Escuela Politécnica Superior

IA para estimación de la tolerancia a fallos en aplicaciones aeroespaciales

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
David Ruiz Falcó

Tutor/es:
Sergio Antonio Cuenca Asensi

Antonio Martínez Álvarez

Universitat d'Alacant
Universidad de Alicante

Septiembre 2019

IA para la estimación de la tolerancia a fallos en aplicaciones aeroespaciales

Autor
David Ruiz Falcó

Directores
Sergio Antonio Cuenca Asensi
Tecnología informática y computación
Antonio Martínez Álvarez
Tecnología informática y computación

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

ALICANTE, 2 de septiembre de 2019

Justificación y Objetivos

La tolerancia a fallos, es decir la capacidad de trabajar de forma correcta aún en presencia de fallos, es un requisito fundamental en los sistemas que operan en entornos afectados por la radiación natural (naves espaciales, satélites, aviones, drones, etc). Evaluar la fiabilidad de estas aplicaciones es una tarea muy costosa que requiere una enorme cantidad de tiempo de cómputo para la simulación de millones de fallos.

El propósito de este proyecto es reducir este problema mediante el desarrollo de un sistema de Inteligencia Artificial. Esta IA debe realizar una estimación a priori de la cobertura frente a fallos a partir del análisis de diversos parámetros de los programas.

Índice general

1 Introducción **1**
- 1.1 Estado del arte . . . 1
- 1.2 Definiciones . . . 1
  - 1.2.1 Tiempo de vida de un registro . . . 1
  - 1.2.2 Accesos a memoria . . . 3
  - 1.2.3 Secciones del ejecutable . . . 4
  - 1.2.4 Tolerancia frente a fallos . . . 5
- 1.3 Objetivo . . . 5

2 Tecnologías **6**
- 2.1 OVPSim . . . 6
- 2.2 Google Drive . . . 6
  - 2.2.1 Google Colab . . . 7
- 2.3 Librerías utilizadas . . . 7

3 Obtención de datos **9**
- 3.1 Introducción . . . 9
- 3.2 Traza de ejecución . . . 9
- 3.3 Tamaño de las cabeceras . . . 13
- 3.4 Tolerancia a fallos . . . 13

4 Procesado de datos **15**
- 4.1 Cálculo tiempo de vida de los registros . . . 15
- 4.2 Cálculo accesos a memoria . . . 23
- 4.3 Procesamiento del conjunto de datos . . . 25

5 Modelo predictivo **31**
- 5.1 Introducción . . . 31
- 5.2 Lectura de datos . . . 32
- 5.3 Estructura del entrenamiento y de la evaluación del modelo . . . 34
- 5.4 Validación cruzada . . . 35
  - 5.4.1 Búsqueda de parámetros óptimos . . . 36
- 5.5 Modelo final . . . 51
- 5.6 Evaluación del modelo . . . 52
- 5.7 Otras pruebas . . . 53
  - 5.7.1 Evaluando el modelo con otro tipo de datos . . . 53

5.7.2 ¿Qué información es la más importante para predecir la tolerancia a fallos? 54

6 Conclusiones **57**

Bibliografía **59**

Índice de figuras

1.1. Ejemplo tiempo de vida registro r1 2
1.2. Ejemplo tiempo de vida registro r2 2
1.3. Ejemplo alteraciones registro r2 3
1.4. Ejemplo tamaños secciones bubblesort.elf 4

2.1. OVP logo 6
2.2. Google Drive logo 7
2.3. Google Colab logo 7

3.1. Ejemplo de una parte de la traza de ejecución 10
3.2. Ejemplo de la traza procesada en formato csv 11
3.3. Fichero de ejemplo simplificado con los resultados de la campaña de inyección de fallos 14

4.1. Sintaxis instrucciones STM y LDM 16
4.2. Tiempos de vida de los registros en uno de los programas 19
4.3. Formato del registro cpsr 22
4.4. Carpetas con la información 25
4.5. Subcarpetas 26
4.6. Ficheros dentro de cada subcarpeta 27
4.7. Identificador de la carpeta 28

5.1. Estructura red neuronal 31
5.2. Neurona 32
5.3. Compartir fichero 33
5.4. Esquema para la obtención del modelo final 35
5.5. Validación cruzada con k=4 36
5.6. 1000 épocas, validación cruzada con k=10 50
5.7. Misma gráfica, mostrando desde la época 100 a la 800 51

Índice de tablas

3.1. Dataframe de pandas con la información de la traza 12

4.1. Relación entre los sufijos condicionales y valores de los flags 21
4.2. Ejemplo datos obtenidos (parte 1) 30
4.3. Ejemplo datos obtenidos (parte 2) 30

5.1. Parámetros escogidos para el modelo 51
5.2. Comparativa predicciones (SDC) 52
5.3. Comparativa predicciones (hang) 53

1 Introducción

1.1. Estado del arte

En la actualidad una de las formas de evaluar la tolerancia a fallos de una aplicación es mediante campañas de inyección de fallos. Estas campañas se basan en insertar fallos reales o simulados en el sistema y comparar los resultados de la ejecución de la aplicación con los que se obtendrían de una ejecución sin fallos. Las técnicas de inyección de fallos se pueden agrupar en dos grupos: basadas en hardware o basadas en software [1].

- Técnicas hardware: la inyección física de fallos utiliza radiación o rayos láser para inducir fallos en circuitos integrados. También se pueden realizar inyecciones lógicas de fallos accediendo a los elementos lógicos del sistema. Esta técnica permite acceder a zonas de difícil acceso y no requiere de ninguna modificación del sistema para la inyección de los fallos, sin embargo puede producir daños físicos en el hardware y presenta una baja portabilidad y observabilidad.

- Técnicas software: para inyectar fallos por software se requiere añadir instrucciones adicionales al programa original. Al no requerir ningún hardware especial esta técnica es más sencilla y presenta un coste menor.

1.2. Definiciones

Se van a explicar algunas definiciones que son importantes para entender este trabajo.

1.2.1. Tiempo de vida de un registro

Se podría definir el tiempo de vida de un registro como el tiempo que un registro permanece con un valor crítico, es decir, que si se modificase el valor del registro durante ese tiempo provocaría que posteriormente se leyera un valor erróneo.

Por ejemplo, en un supuesto programa que se han ejecutado 10 instrucciones, en el cual el registro r1 ha sido escrito en la instrucción 3 y ha sido leído en la instrucción 7, r1 tendría un tiempo de vida de 4 instrucciones:

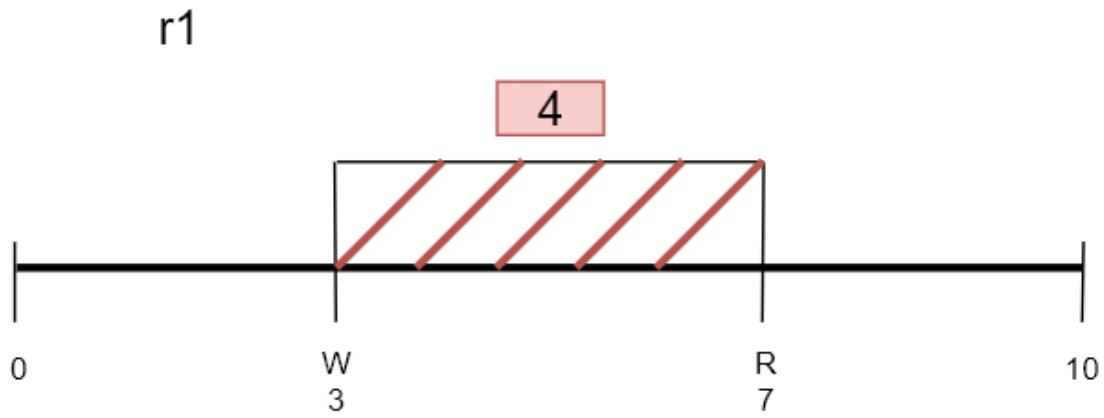

Figura 1.1: Ejemplo tiempo de vida registro r1

Otro ejemplo para el registro r2, en el supuesto programa, donde r2 es leido por las instrucciones 2, 8 y 9 y escrito por las instrucciones 4 y 6:

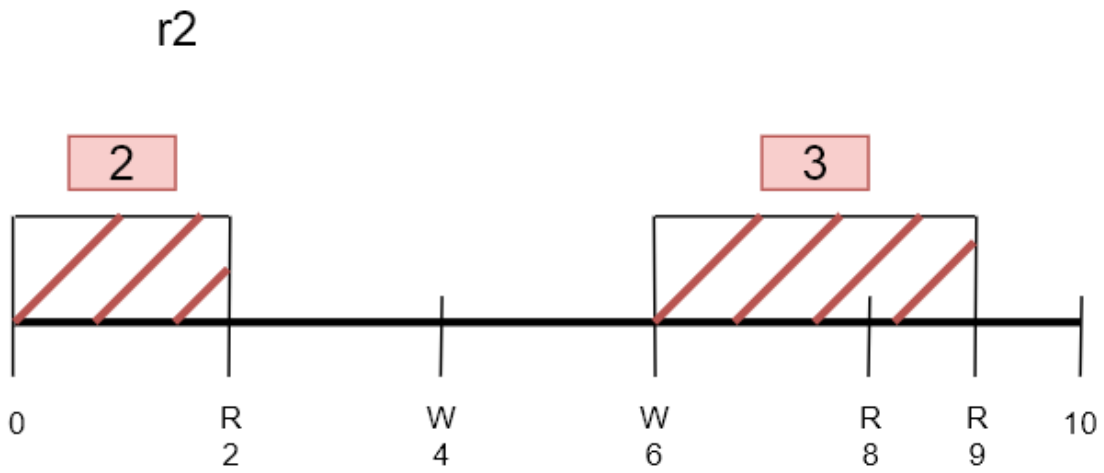

Figura 1.2: Ejemplo tiempo de vida registro r2

En el ejemplo de arriba el tiempo de vida del registro r2 sería de 5.

Si durante la ejecución de las instrucciones 0 a la 2 o de la 6 a la 9 se produjese una alteración del registro r2 (por ejemplo debido a la radiación), esta alteración podría derivar en un fallo en la ejecución del programa ya que se leería un valor erróneo en las instrucciones de lectura de r2. A diferencia de si esta alteración del registro hubiese

ocurrido durante la ejecución de las instrucciones 2 a la 6 o de la 9 hasta la 10 que no hubiese pasado nada.

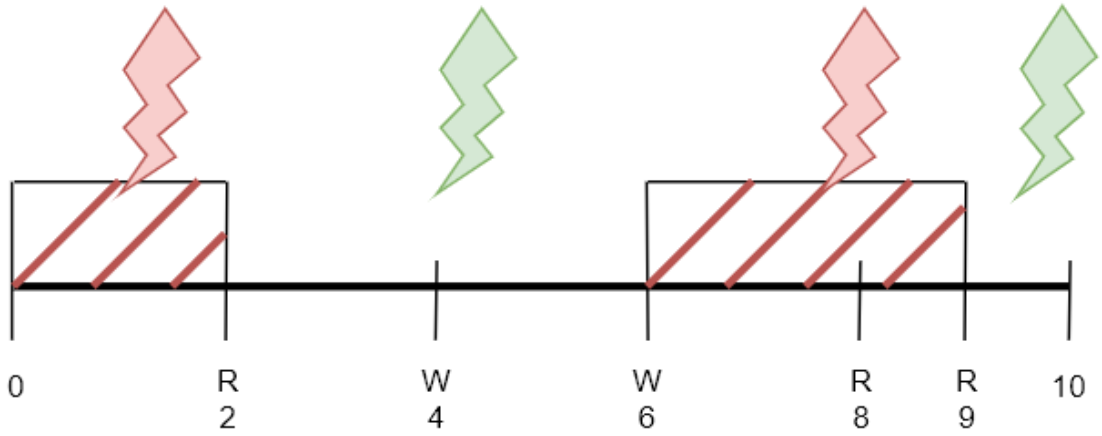

Figura 1.3: Ejemplo alteraciones registro r2

En la imagen de arriba las flechas verdes corresponderían a alteraciones del valor del registro que no provocan ningún fallo y las flechas rojas a alteraciones que podrían provocar algún fallo.

1.2.2. Accesos a memoria

Los accesos a memoria como su nombre indica son instrucciones que acceden a memoria. Estos accesos pueden ser de lectura (si leen de la memoria) o de escritura (si escriben en la memoria).

Por ejemplo en un programa en el que se ejecuten las siguientes instrucciones:

```
1 ldr r0,[r3]
2 mov r0, r4
3 str r4,[r3]
```

La instrucción ldr carga en r0 el valor que hay en la dirección de memoria r3 por lo que accede a memoria para leer, la instrucción mov copia el valor de r4 a r0 por lo que no accede a memoria y la instrucción str almacena el valor de r4 en la dirección de memoria r3 por lo que accede a memoria para escribir.

En total tendríamos dos accesos a memoria, uno para leer y otro para escribir.

1.2.3. Secciones del ejecutable

Un programa en arm está compuesto por varias secciones, cada uno con su respectivo tamaño. Estos tamaños se pueden ver utilizando el comando de linux *objdump -h <ejecutable>*.

Salida al ejecutar el comando *objdump -h bubblesort.elf*:

```
bubblesort.ARM.elf:     file format elf32-little

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .note.gnu.build-id 00000024  00008000  00008000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .init         00000018  00008024  00008024  00008024  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text         00002868  00008040  00008040  00008040  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .fini         00000018  0000a8a8  0000a8a8  0000a8a8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .rodata       00000328  0000a8c0  0000a8c0  0000a8c0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .ARM.exidx    00000008  0000abe8  0000abe8  0000abe8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000004  0000abf0  0000abf0  0000abf0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .init_array   00000004  0001abf4  0001abf4  0000abf4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 .fini_array   00000004  0001abf8  0001abf8  0000abf8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .data         00000958  0001ac00  0001ac00  0000ac00  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 10 .bss          0000029c  0001b558  0001b558  0000b558  2**2
                  ALLOC
 11 .comment      0000002d  00000000  00000000  0000b558  2**0
                  CONTENTS, READONLY
 12 .debug_aranges 00000338  00000000  00000000  0000b588  2**3
                  CONTENTS, READONLY, DEBUGGING
 13 .debug_info   000107ef  00000000  00000000  0000b8c0  2**0
                  CONTENTS, READONLY, DEBUGGING
 14 .debug_abbrev 00003667  00000000  00000000  0001c0af  2**0
                  CONTENTS, READONLY, DEBUGGING
 15 .debug_line   000056a3  00000000  00000000  0001f716  2**0
                  CONTENTS, READONLY, DEBUGGING
 16 .debug_frame  000009a4  00000000  00000000  00024dbc  2**2
                  CONTENTS, READONLY, DEBUGGING
 17 .debug_str    0000257d  00000000  00000000  00025760  2**0
                  CONTENTS, READONLY, DEBUGGING
 18 .debug_loc    00003680  00000000  00000000  00027cdd  2**0
                  CONTENTS, READONLY, DEBUGGING
 19 .debug_ranges 00000588  00000000  00000000  0002b35d  2**0
                  CONTENTS, READONLY, DEBUGGING
 20 .ARM.attributes 0000002f  00000000  00000000  0002b8e5  2**0
                  CONTENTS, READONLY
```

Figura 1.4: Ejemplo tamaños secciones bubblesort.elf

Como se puede ver en la imagen superior este ejecutable está compuesto por 20 secciones aunque para este trabajo solo se usarán cuatro:

- .text: contiene el código y los datos que son constantes.
- .data: contiene los datos inicializados.
- .rodata: contiene constantes estáticas.
- .bss: contiene los datos no inicializados.

La información que se utilizarán son los tamaños de las anteriores secciones (text, data, rodata, bss) que están en la tercera columna de la imagen anterior y en valor hexadecimal.

1.2.4. Tolerancia frente a fallos

La tolerancia frente a fallos es la capacidad de un programa de seguir su ejecución correctamente después de haberse producido un fallo.

Cuando se produce un fallo puede suceder lo siguiente:

- El programa se ejecuta correctamente (unAce).
- Se produce algún error en el programa (Ace), pudiendo ocurrir dos cosas:
  - El programa termina, pero devolviendo un valor erróneo (SDC: Silent Data Corruption).
  - El programa no termina, se queda colgado (Hang).

1.3. Objetivo

El objetivo de este trabajo es a partir de información obtenida de un programa (tiempo de vida de los registros, accesos a memoria y tamaño de las secciones) predecir la tolerancia a fallos del programa (predecir porcentaje de unAce, SDC y Hang).

2 Tecnologías

2.1. OVPSim

OVPsim [2] es un simulador de sistema completo que se utiliza para ejecutar binarios diseñados para un hardware distinto al de origen. Se ha utilizado para obtener el fichero de las trazas de ejecución del ejecutable, con el que posteriormente calcularemos el tiempo de vida de los registros y los accesos a memoria. También se ha utilizado para realizar las campañas de inyección de fallos mediante un plugin para OVPSim desarrollado por un compañero del grupo de investigación [3].

Figura 2.1: OVP logo

2.2. Google Drive

Google Drive es desde donde se ha realizado casi todo el trabajo. Ahí se han almacenado todos los ficheros de datos (trazas de ejecución, ejecutables, valores de tolerancia a fallos, etc.) y también los cuadernos de python. En total se han utilizado 118,2 GB de espacio en Google Drive.

Figura 2.2: Google Drive logo

2.2.1. Google Colab

Dentro de Google Drive hay diversas extensiones. Una de ellas es Google Colab [4]. Esta herramienta permite programar en python desde Google Drive y ejecutar el código en un ordenador en la nube por lo que lo único que se necesita es conexión a internet.

Figura 2.3: Google Colab logo

2.3. Librerías utilizadas

He utilizado varías librerías de python:

- Pydrive [5]: utilizado tanto para descargar como para subir ficheros a Google Drive con python. Esto ha sido muy útil al utilizarlo junto a Google Colab ya que con Google Colab yo solo tenía acceso a la máquina en la nube a través del cuaderno de python y como toda la información con la que se ha trabajado estaba subida a Google Drive necesitaba una herramienta para envíar la información desde Google Drive a la máquina en la nube que tenía asignada

- Keras [6]: utilizado para crear redes neuronales, entrenarlas y evaluarlas para obtener finalmente un modelo predictivo.

- Scikit-learn [7]: principalmente lo he utilizado para separar los datos en distintos conjuntos (entrenamiento, validación y test), para la validación cruzada y para la automatización de la optimización de los hiperparámetros del modelo.

- re [8]: librería que permite declarar expresiones regulares y realizar operaciones con ellas. Las expresiones regulares permiten realizar búsquedas en cadenas de caracteres. Por ejemplo la expresión regular *"\s{8}abc"* permitiría buscar cadenas que contienen 8 espacios en blanco seguido de las letras a, b y c en ese orden.

- Pandas [9]: librería que se utiliza para la manipulación y análisis de datos, se ha utilizado para leer los ficheros en formato csv.

3 Obtención de datos

3.1. Introducción

Esta es la parte más importante, ya que es donde se obtendrán los datos con los que se entrenará nuestro modelo, cualquier error cometido en la obtención de los datos afectará a la capacidad de aprendizaje del modelo.

La información recopilada para cada programa es:

- Los tiempos de vida de los registros y los accesos a memoria, utilizando la traza de ejecución (obtenida con OVPSim).
- Los tamaños de las cabeceras (desensamblando el ejecutable).
- La tolerancia a fallos del ejecutable (mediante una campaña de inyección de fallos, también con OVPSim).

3.2. Traza de ejecución

La traza de ejecución muestra las instrucciones que se han ido ejecutando y también los valores que tienen los registros después de cada instrucción.

```
Warning (DWR_VER) Unsupported Dwarf version 4 found - only version 3 and earlier supported
Info 1: 'BareMetalArmCortexASingle/cpu0', 0x0000000000008134(_start): e3b00016 movs    r0,#22
Info   r0 00000000 -> 00000016
Info 'BareMetalArmCortexASingle/cpu0' REGISTERS
        r0      0x16        22
        r1      0x0         0
        r2      0x0         0
        r3      0x0         0
        r4      0x0         0
        r5      0x0         0
        r6      0x0         0
        r7      0x0         0
        r8      0x0         0
        r9      0x0         0
        r10     0x0         0
        r11     0x0         0
        r12     0x0         0
        sp      0x0         0x0
        lr      0x0         0
        pc      0x8138      0x8138
        fps     0x0         0
        cpsr    0x1d3       467
```

Figura 3.1: Ejemplo de una parte de la traza de ejecución

En la imagen se ve una parte de la traza con toda la información que se obtiene para cada instrucción. Para la obtención de los tiempos de vida de los registros y de los accesos a memoria solo hacen falta la instrucción y el valor del registro cpsr (ambos rodeados en rojo en la imagen).

El motivo por el que hace falta el valor del registro cpsr es porque este registro contiene los valores de los flags (N, Z, C, V) necesarios para saber si se han ejecutado las instrucciones condicionales (por ejemplo las instrucciones addeq o ldrne).

Para procesar el fichero se utilizan dos expresiones regulares, una que detecta la linea donde está la instrucción y otra que detecta la linea donde esta el registro cpsr.

Como se puede ver en la imagen anterior, la linea donde esta la instrucción comienza por 'Info 1', la siguiente instrucción comenzaría por 'Info 2' y así sucesivamente, por lo que usando la expresión regular 'Info\s[0-9]' detectaríamos las lineas donde están las instrucciones. E igualmente para detectar la linea del registro cpsr se utilizaría:'\s{8}cpsr'.

Una vez se tienen las expresiones regulares que detectan las lineas se pueden guardar fácilmente los datos que nos interesan dividiendo las lineas con la función split que por defecto las divide eligiendo como delimitador los espacios en blanco. Por ejemplo de la linea 'cpsr 0x1d3 467' se obtendría el valor 467 con el siguiente código:

```
1 if re.match(r"^\s{8}cpsr",line): #Si la linea cumple la
      expresión regular
2     columns = line.split() #Se divide la linea
3     fileToWrite.write('%s\n' % (columns[2])) #Se escribe el valor
          número
```

Para leer los datos (instrucciones y valor cpsr) con python se utiliza la libraría pandas pero antes se pasan esos datos a formato csv con el siguiente código:

```
instrLine = r"^Info\s[0-9]"
cpsrLine = r"^\s{8}cpsr"
fileToWrite = open("traceFile.csv", "w")
with open("trace.log") as file:
  for line in file:
    if re.match(instrLine,line):
      columns = line.split()
      fileToWrite.write('%s;' % (columns[5]))
      if(len(columns)>6):
        registersAB = columns[6].split(",",1)
        if(len(registersAB)==1):
          fileToWrite.write('%s;%s;' % (registersAB[0], "-"))
        else:
          fileToWrite.write('%s;%s;' % (registersAB[0],
             registersAB[1]))
    elif re.match(cpsrLine,line):
      columns = line.split()
      fileToWrite.write('%s\n' % (columns[2]))
fileToWrite.close()
```

Con el código anterior se procesa el fichero de la figura 3.2 dejándolo con el siguiente formato:

```
movs;r0;#22;467
add;r1;pc,#272;467
svc;#0x123456;-;467
ldr;r0;[pc,#264];467
ldr;r1;[r0];467
cmp;r1;#0;1610613203
bne;8158;-;1610613203
ldr;r1;[pc,#272];1610613203
```

Figura 3.2: Ejemplo de la traza procesada en formato csv

En el formato csv cada elemento se separa con un punto y coma. El fichero se divide en tantas líneas como instrucciones se hayan ejecutado. Cada línea se divide en 4 columnas:

- La primera columna es la instrucción ejecutada
- La segunda columna es el primer parámetro de la instrucción.

- La tercera columna es el segundo parámetro de la instrucción, en el caso de que la instrucción no tenga un tercer parámetro (por ejemplo la instrucción bne 8158) se escribe un guión.

- La cuarta columna es el valor del registro cpsr.

La forma de dividir los parámetros de una instrucción es separándolos por la primera coma que se encuentre. Por ejemplo la instrucción *add r1,pc,#272* se separaría en add;r1;pc,#272.

Una vez se tiene el fichero en formato csv se puede leer fácilmente con la librería pandas:

```
df = pandas.read_csv("traceFile.csv", sep=";", names=["Instruction
    ","A","B","cpsr"])
```

	Instruction	**A**	**B**	**cpsr**
0	movs	r0	#22	0
1	add	r1	pc,#272	467
2	svc	#0x123456	-	467
3	ldr	r0	[pc,#264]	467
4	ldr	r1	[r0]	467
5	cmp	r1	#0	467
6	bne	8158	-	1610613203
7	ldr	r1	[pc,#272]	1610613203
8	str	r1	[r0]	1610613203
9	ldr	r1	[r0,#8]	1610613203
...	...	...	...	...
57927	ldm	sp!	{r4,r5,r6,r7,r8,r9,sl,fp,pc}	1610613203
57928	movw	r3	#43936	1610613203
57929	movt	r3	#0	1610613203
57930	ldr	r0	[r3]	1610613203
57931	ldr	r3	[r0,#60]	1610613203
57932	cmp	r3	#0	1610613203
57933	beq	8394	-	1610613203
57934	mov	r0	r4	1610613203
57935	bl	8bd8	-	1610613203
57936	INTERCEPT	***	-	1610613203

Tabla 3.1: Dataframe de pandas con la información de la traza

Con esto ya se tendría la traza filtrada y leída. En el apartado de procesado de datos se explicará como obtener el tiempo de vida de los registros y los accesos a memoria a

partir de esta información.

3.3. Tamaño de las cabeceras

La siguiente información que se va a obtener es el tamaño de las secciones .text, .data, .bss y .rodata del ejecutable.

Como se puede ver en la figura 1.4 el formato del fichero es muy sencillo, para encontrar las lineas donde se encuentran el tamaño de las secciones que se van a guardar se utilizan las siguientes expresiones regulares:

```
textLine = r"^.*\.text"
dataLine = r"^.*\.data"
bssLine = r"^.*\.bss"
rodataLine = r"^.*\.rodata"
```

En el siguiente código vemos la función que lee el fichero linea a linea, y cuando una linea coincide con la expresión regular vista anteriormente, divide la linea y se queda con el tercer elemento (el tamaño de la sección), el cual se convierte a decimal antes de guardarlo en la variable:

```
def getSizesProgram():
  with open("objdump.txt") as file:
    for line in file:
      if re.match(textLine, line):
        text = int(line.split()[2], 16)
      if re.match(dataLine, line):
        data = int(line.split()[2], 16)
      if re.match(bssLine, line):
        bss = int(line.split()[2], 16)
      if re.match(rodataLine, line):
        rodata = int(line.split()[2], 16)
  return text, data, bss, rodata
```

3.4. Tolerancia a fallos

La tolerancia a fallos, como se ha comentado antes, se ha obtenido mediante un plugin de OVPSim que permite realizar una campaña de inyección de fallos sobre un programa y que posteriormente te devuelve un fichero con los resultados.

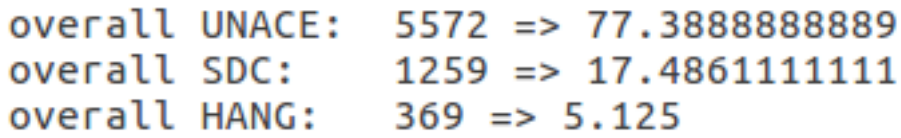

Figura 3.3: Fichero de ejemplo simplificado con los resultados de la campaña de inyección de fallos

El fichero que se devuelve contiene bastante más información, pero lo importante son las tres lineas que contiene los valores generales de unAce, SDC y Hang. De la imagen anterior podemos ver que en el programa cuando se produce alguna anomalía en algún valor tiene un 77.38 % de probabilidades de que no se produzca ningún fallo (unAce), un 17.48 % de que se produzca un fallo de tipo SDC y un 5.12 % de que se produzca un fallo de tipo Hang.

Y para obtener la información de este fichero, al igual que en los ficheros anteriores, declaramos las expresiones regulares para detectar las lineas donde está la información, en este caso son tres expresiones regulares. Luego dividimos la linea con la función split y el quinto elemento sería el elemento que queremos guardar. Esto lo hace la siguiente función de python:

```
1  unaceLine = r"^overall\sUNACE"
2  sdcLine = r"^overall\sSDC"
3  hangLine = r"^overall\sHANG"
4
5  def getCoverage():
6    with open("coverage.txt") as file:
7      for line in file:
8        if re.match(unaceLine, line):
9          unace = line.split()[4]
10       if re.match(sdcLine, line):
11         sdc = line.split()[4]
12       if re.match(hangLine, line):
13         hang = line.split()[4]
14   return unace, sdc, hang
```

4 Procesado de datos

4.1. Cálculo tiempo de vida de los registros

Una vez se tiene leída la traza de ejecución en python usando pandas (tabla 3.1) se pueden calcular los tiempos de vida de los registros. Como se puede ver en la tabla 3.1 cada instrucción tiene dos parámetros A y B cada uno con sus valores correspondientes. Hay que dividir las instrucciones en tres grupos según se escriban o se lean los registros de cada parámetro:

- Grupo 1: Se leen los registros del parámetro A y se escriben los del parámetro B. Ejemplo de instrucciones que pertenecen a este grupo: *ldm r8,{r0,r2,r9}*. En la instrucción anterior se lee el registro r8 y se escribe en los registros r0, r2 y r9.

- Grupo 2: Se escribe en los registros del parámetro A y se leen los del parámetro B. Ejemplo de instrucciones que pertenecen a este grupo: *mov r11,r3* o *add r2,r3,r4*. En las instrucciones anteriores se leen los registros r3 y r4 y se escribe en los registros r11 y r2.

- Grupo 3: Se leen los registros del parámetro A y también se leen los del parámetro B. Ejemplo de instrucciones que pertenecen a este grupo: *str r2,[r2, #0x100]* o *cmp r2,r9*. En las instrucciones anteriores se leen los registros r2 y r9.

Para ver a que grupo pertenece cada instrucción se utilizan expresiones regulares.

Primero se declaran unos sufijos que pueden tener algunas instrucciones:

```
1 arm_cond = r"(eq|ne|cs|cc|mi|pl|vs|vc|hi|ls|ge|lt|gt|le){0,1}$"
2 arm_type = r"(d|b|sb|h|sh){0,1}"
3 addr_mode = r"(ia|ib|da|db|fd|fa|ed|ea){0,1}"
4 sflag = r"s{0,1}"
```

La expresión regular que detecta instrucciones pertenecientes al primer grupo es:

```
1 ldm = r"^ldm"+addr_mode+arm_cond
```

En este caso solo habría una instrucción la cual puede contener el sufijo addr_mode y/o arm_cond que se han declarado anteriormente.

Para saber que sufijos pueden tener las instrucciones, en la documentación de arm [10] viene esta información. Por ejemplo si vemos la instrucción ldm en la documentación de arm:

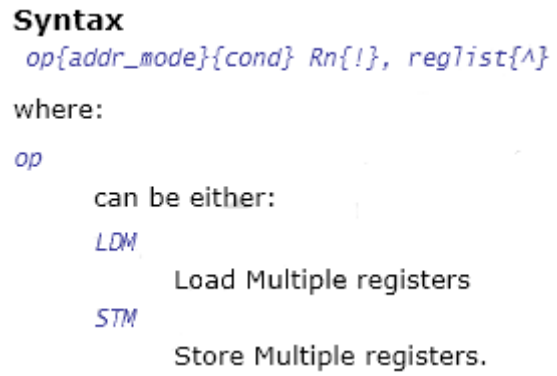

Figura 4.1: Sintaxis instrucciones STM y LDM

En este caso ldm y stm comparten sintaxis y ambos pueden tener como sufijos addr_mode y/o arm_cond.

Las expresiones regulares que detectan instrucciones pertenecientes al segundo grupo son:

```
mov = r"^(mov|movw|movt|mvn)"+sflag+arm_cond
op = r"^(add|sub|mul|mla|rsb)"+sflag+arm_cond
ldr = r"^ldr"+arm_type+arm_cond
lsr = r"^(asr|lsl|lsr|ror|rrx)"+sflag+arm_cond
uxtb = r"^uxtb"+arm_cond
logical = r"^(and|orr|eor|bic|orn)"+sflag+arm_cond
```

En este caso al haber varias instrucciones pertenecientes a este grupo hacen falta varias expresiones regulares cada una siguiendo una sintaxis que se puede ver en la documentación de arm. Estas expresiones regulares se pueden agrupar en una sola expresión quedando así:

```
writeReadInstr = r"^("+mov+"|"+op+"|"+ldr+"|"+lsr+"|"+uxtb+"|"+
    logical+")$"
```

Las expresiones regulares que detectan instrucciones pertenecientes al tercer grupo son:

```
str1 = r"^str"+arm_type+arm_cond
stm = r"^stm"+addr_mode+arm_cond
branch = r"^(b|bl|bx|blx)"+arm_cond
testop = r"^(tst|teq)"+arm_cond
compare = r"^(cmp|cmn)"+arm_cond
pld = r"^(pld|pldw|pli)"+arm_cond
```

E igualmente, agrupándolas en una sola expresión quedaría así:

```
readReadInstr = r"^("+str1+"|"+stm+"|"+branch+"|"+testop+"|"+
    compare+"|"+pld+")$"
```

Hay que mencionar que solo se han clasificado en estos tres grupos las instrucciones que aparecen en las trazas que se han utilizado, en arm hay muchas más instrucciones pero al no aparecer en las trazas que se han usado no se han incluido en las expresiones regulares.

Una vez se tienen las expresiones regulares, el algoritmo que calcula el tiempo de vida de los registros se basa en el siguiente código:

```
registers = ["r0","r1","r2","r3","r4","r5","r6","r7","r8","r9","sl
    ","fp","ip","sp","lr"]

lifetime = {}
values = {}
for i in registers:
  values[i] = 0
  lifetime[i] = 0

def calculateLTRegRead(read):
  for reg in registers:
    if reg in read:
      #index es la posicion de la instruccion actual
      lifetime[reg]+=index-values[reg]
      values[reg]=index

def calculateLTRegWritten(written):
  for reg in registers:
    if reg in written:
      values[reg]=index
```

La lista 'registers' contiene los nombres de los registros de los que queremos calcular su tiempo de vida, en este caso hay 15 registros. Primero se crean dos diccionarios ('lifetime' y 'values'), ambos contendrán como clave los nombres de los registros y estarán inicializados a cero. El diccionario 'lifetime' se utilizará para guardar el valor del tiempo de vida de cada registro mientras que el diccionario 'values' se usará para almacenar la posición de la última instrucción que ha leído o escrito cada registro. Las dos funciones que hay al final del código son muy parecidas, la primera es la que recibe los parámetros de la instrucción que son leidos y la segunda los que son escritos, por ejemplo al leer la instrucción *ldm r8,{r0,r2,r9}* se le pasaría a la primera función la cadena 'r8' y a la segunda función la cadena '{r0,r2,r9}'. Lo que hace la primera función es recorrer todos los registros, cuando uno de estos registros se encuentra en la cadena que se le pasa a la función (en el ejemplo anterior sería 'r8'), actualiza el tiempo de vida de ese registro añadiéndole la diferencia entre la posición de la instrucción actual menos la última posición de la instrucción que leyó o escribió en ese registro. Luego actualizaría el registro del diccionario values con la posición de la instrucción actual.

La segunda función actualiza el diccionario 'values' de cada registro que es leído, en el ejemplo anterior para los registros r0, r2 y r9, con el valor de la posición de la instrucción actual.

Hay que juntar esto último con las expresiones regulares y un bucle que vaya leyendo la traza (tabla 3.1) linea a linea:

```
def getInformationProgram():

  def calculateLTRegRead(read):
    for reg in registers:
      if reg in read:
        lifetime[reg]+=index-values[reg]
        values[reg]=index

  def calculateLTRegWritten(written):
    for reg in registers:
      if reg in written:
        values[reg]=index

  lifetime = {}
  values = {}
  for i in registers:
    values[i] = 0
    lifetime[i] = 0

  index = 0
  for row in df.itertuples(index=True, name="line"):
    instr = row[1]
    index = row[0]
```

```
24      if(re.match(writeReadInstr,instr)): # Write Read
25        calculateLTRegRead(row[3]) #row[3] = parámetro B
26        calculateLTRegWritten(row[2]) #row[2] = parámetro A
27      elif(re.match(readReadInstr,instr)): # Read Read
28        calculateLTRegRead(row[2])
29        calculateLTRegRead(row[3])
30      elif(re.match(ldm,instr)): # Read Write (ldm)
31        calculateLTRegRead(row[2])
32        calculateLTRegWritten(row[3])
33
34    return lifetime
```

Como se puede observar en el código anterior se ha añadido un bucle for que recorre la traza y se han añadido tres condiciones que comprueban a que grupo pertenece la instrucción utilizando las expresiones regulares anteriores. Esta función devolvería el tiempo de vida de los registros.

Un ejemplo de la información devuelta:

	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	sl	fp	ip	sp	lr
LifeTime	11298	15732	56032	57175	57889	57868	57198	57952	57956	57968	57981	57945	56983	58010	57011

Figura 4.2: Tiempos de vida de los registros en uno de los programas

Cabe mencionar que por ejemplo en la primera condición del código anterior, se llama primero a la función 'calculateLTRegRead' para que no den problemas las instrucciones que escriben y leen en un mismo registro (ej. *add r2,r2,r3*).

Por último solo faltaría tener en cuenta las instrucciones condicionales ya que instrucciones como *addeq r1,r2,r3* podrían no ejecutarse dependiendo de los flags.

Primero declaramos la expresión regular que detecta instrucciones condicionales:

```
1  doInstr = r"(eq|ne|cs|cc|mi|pl|vs|vc|hi|ls|ge|lt|gt|le)$"
```

Y añadimos en una lista algunas instrucciones que producían falsos positivos porque se detectaban como instrucciones condicionales cuando no lo eran:

```
1  exInstr = ["teq","movs","svc","lsls","bics","muls"]
```

Ahora añadiéndo esta condición al bucle anterior se puede saltar una iteración del bucle si la instrucción condicional no se ejecuta.

```
condMatch = re.search(doInstr,instr)
if(condMatch and (instr not in exInstr)):
      if not executedInstr(condMatch.group(0), row[4]):
        continue
```

Quedando el bucle así:

```
for row in df.itertuples(index=True, name="line"):
    instr = row[1]
    index = row[0]
    condMatch = re.search(doInstr,instr)
    if(condMatch and (instr not in exInstr)):
      setCond.add(instr)
      if not executedInstr(condMatch.group(0), row[4]):
        continue
    if(re.match(writeReadInstr,instr)): # Write Read
      calculateLTRegRead(row[3])
      calculateLTRegWritten(row[2])
    elif(re.match(readReadInstr,instr)): # Read Read
      calculateLTRegRead(row[2])
      calculateLTRegRead(row[3])
    elif(re.match(ldm,instr)): # Read Write (ldm)
      calculateLTRegRead(row[2])
      calculateLTRegWritten(row[3])
```

La función 'executedInstr' recibe el sufijo condicional y el valor cpsr y devuelve verdadero si la instrucción ha sido ejecutada y falso en caso contrario.

La forma de saber si una instrucción es ejecutada es obteniendo los flags del registro cpsr y compararlo con el sufijo condicional usando la siguiente tabla de la documentación de arm:

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS	C = 1	Higher or same, unsigned
CC	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed

Tabla 4.1: Relación entre los sufijos condicionales y valores de los flags

Para sacar los valores de los cuatro flags del valor del registro cpsr hay que pasar el valor decimal del cpsr a binario y los cuatro bit más significativos son los valores de cada uno de los flags:

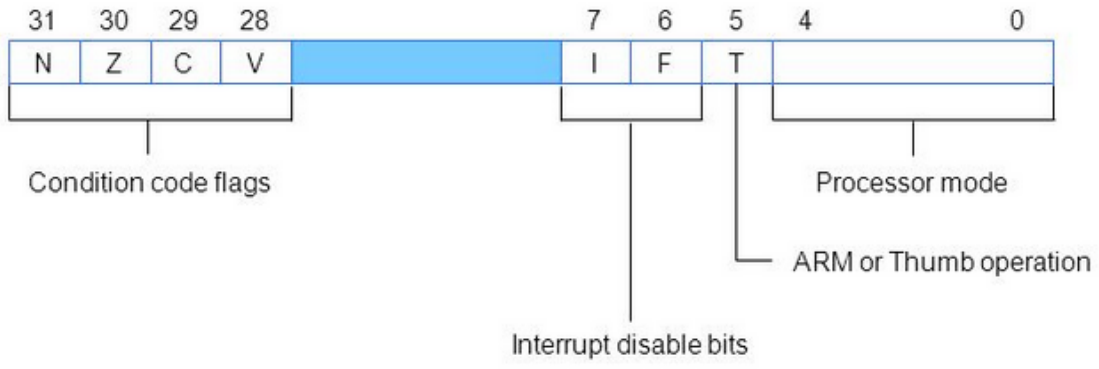

Figura 4.3: Formato del registro cpsr

La función 'executedInstr' que hace todo esto sería:

```
def executedInstr(cond, cpsr):
  binarynum = bin(cpsr)[2:] #convertimos el valor cpsr a binario

  #las excepciones son porque al no haber 0s a la izquierda puedes
      intentar acceder a un índice que no existe
  try:
    N = int(binarynum[31])
  except IndexError:
    N = 0
  try:
    Z = int(binarynum[30])
  except IndexError:
    Z = 0
  try:
    C = int(binarynum[29])
  except IndexError:
    C = 0
  try:
    V = int(binarynum[28])
  except IndexError:
    V = 0

  if(cond=="eq"):
    return Z == 1:
  elif(cond=="ne"):
    return Z == 0:
  elif(cond=="cs"):
    return C == 1:
  elif(cond=="cc"):
    return C == 0:
```

```
30    elif(cond=="mi"):
31      return N == 1:
32    elif(cond=="pl"):
33      return N == 0:
34    elif(cond=="vs"):
35      return V == 1:
36    elif(cond=="vc"):
37      return V == 0:
38    elif(cond=="hi"):
39      return (C == 1) and (Z == 0):
40    elif(cond=="ls"):
41      return (C == 0) and (Z == 1):
42    elif(cond=="ge"):
43      return N == V:
44    elif(cond=="lt"):
45      return N != V:
46    elif(cond=="gt"):
47      return(Z == 0) and (N == V):
48    elif(cond=="le"):
49      return (Z == 1) and (N != V)
50    return False
```

Con esto ya se tendría calculado el tiempo de vida de los registros correctamente.

4.2. Cálculo accesos a memoria

Para calcular los accesos a memoria declaramos tres expresiones regulares una para los accesos a memoria totales, otro para los accesos a memoria de lectura y el tercero para los de escritura:

```
1  memoryAccesInstr = r"^("+str1+"|"+stm+"|"+ldr+"|"+ldm+"|"+pld+")$"
2  memoryReadInstr = r"^("+ldr+"|"+ldm+"|"+pld+")$"
3  memoryWriteInstr = r"^("+str1+"|"+stm+")$"
```

Las instrucciones que leen de memoria son ldr, ldm y pld y las que escriben en memoria son str y stm.

Aunque se podrían usar solo dos expresiones regulares ya que $AccesosMemoria = AccesosLectura + AccesosEscritura$, puede ser útil calcular los tres valores para luego comprobar que se han calculado correctamente.

Para obtener los accesos a memoria se pone el siguiente bloque de código dentro del

bucle que recorre la traza de ejecución:

```
if(re.match(memoryAccesInstr,instr)):
      memoryAccess += 1
      if(re.match(memoryReadInstr,instr)):
        memoryRead += 1
      if(re.match(memoryWriteInstr,instr)):
        memoryWrite += 1
```

Quedando finalmente así:

```
def getInformationProgram():

  def calculateLTRegRead(read):
    for reg in registers:
      if reg in read:
        lifetime[reg]+=index-values[reg]
        values[reg]=index

  def calculateLTRegWritten(written):
    for reg in registers:
      if reg in written:
        values[reg]=index

  memoryAccess = 0
  memoryRead = 0
  memoryWrite = 0

  lifetime = {}
  values = {}
  for i in registers:
    values[i] = 0
    lifetime[i] = 0

  index = 0
  for row in df.itertuples(index=True, name="line"):
    instr = row[1]
    index = row[0]
    condMatch = re.search(doInstr,instr)
    if(condMatch and (instr not in exInstr)):
      if not executedInstr(condMatch.group(0), row[4]):
        continue
    if(re.match(writeReadInstr,instr)): # Write Read
      calculateLTRegRead(row[3])
      calculateLTRegWritten(row[2])
    elif(re.match(readReadInstr,instr)): # Read Read
      calculateLTRegRead(row[2])
      calculateLTRegRead(row[3])
```

```
38      elif(re.match(ldm,instr)): # Read Write (ldm)
39        calculateLTRegRead(row[2])
40        calculateLTRegWritten(row[3])
41
42      if(re.match(memoryAccesInstr,instr)):
43        memoryAccess += 1
44        if(re.match(memoryReadInstr,instr)):
45          memoryRead += 1
46        if(re.match(memoryWriteInstr,instr)):
47          memoryWrite += 1
48
49    return lifetime, memoryAccess, memoryRead, memoryWrite
```

4.3. Procesamiento del conjunto de datos

El conjunto de datos con el que se va a trabajar ha sido obtenido por mi compañero del grupo de investigación y está repartido en tres carpetas:

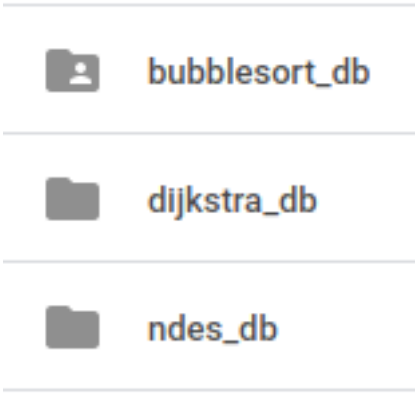

Figura 4.4: Carpetas con la información

Cada carpeta corresponde con un programa distinto:

- Bubblesort: algoritmo de ordenamiento (`https://github.com/mageec/beebs/blob/master/src/bubblesort/libbubblesort.c`).
- Dijkstra: algoritmo para la búsqueda del camino más corto (`https://github.com/mageec/beebs/blob/master/src/dijkstra/dijkstra_small.c`).
- Ndes: algoritmo de cifrado (`https://github.com/mageec/beebs/blob/master/src/ndes/libndes.c`).

Dentro de cada carpeta encontramos otras subcarpetas que corresponden a instancias distintas del mismo programa (el código fuente es el mismo pero cambiaría el código generado por el compilador), por ejemplo dentro de la carpeta bubblesort_db hay 824 subcarpetas y cada una de estas subcarpetas contendrían los ficheros que se necesitan (traza, ejecutable y resultado de la campaña de inyección de fallos).

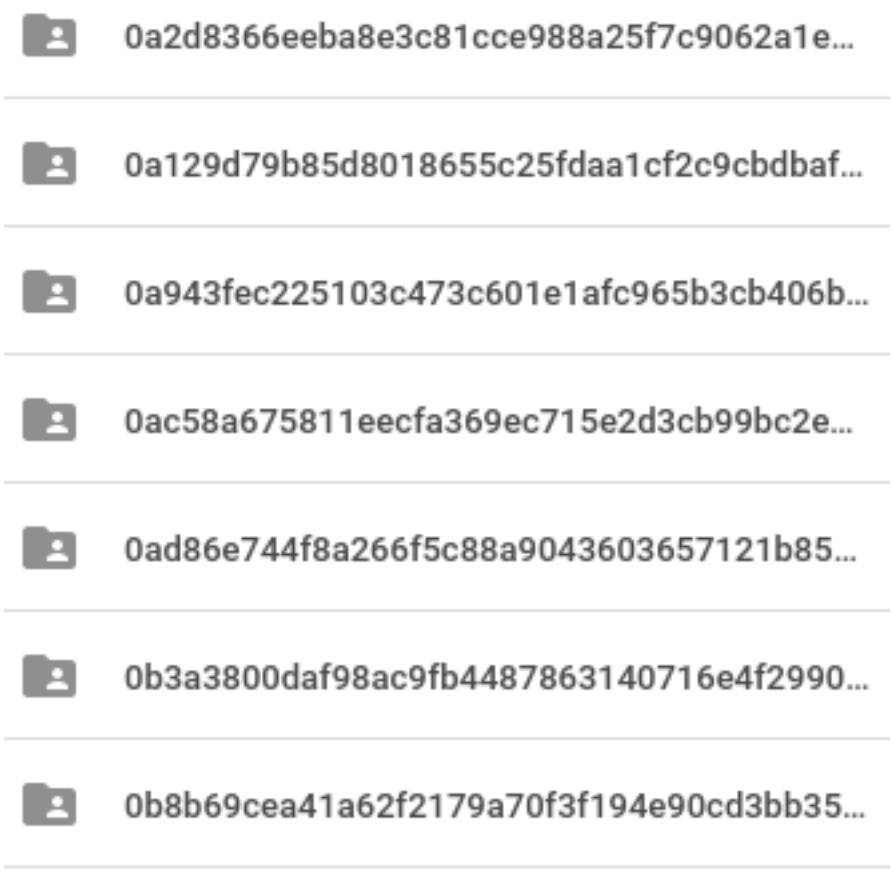

Figura 4.5: Subcarpetas

Dentro de cada subcarpeta hay los siguientes ficheros:

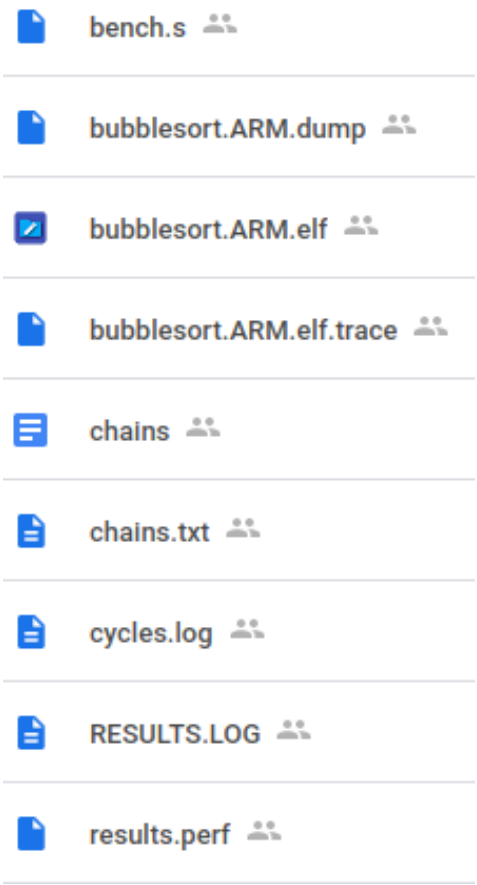

Figura 4.6: Ficheros dentro de cada subcarpeta

Los ficheros que usaremos serán:

- bubblesort.ARM.elf: el ejecutable.
- bubblesort.ARM.elf.trace: la traza de ejecución.
- results.perf: el resultado de la campaña de inyección de fallos.

Todas estas carpetas están subidas a Google Drive por lo que se tendrán que leer desde allí, para ello se usará la librería pydrive.

Primero se importan las librerías necesarias y se otorga permiso al cuaderno de python

para acceder a nuestro Google Drive:

```
import os
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth, files
from oauth2client.client import GoogleCredentials

auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

Luego se crea una lista con todas las subcarpetas con la siguiente función:

```
folder_list = drive.ListFile({'q': "'1SqrpKIGQagZq3AGS5BbNPElJ-
    QF7E8lo' in parents"}).GetList()
```

El identificador '1SqrpKIGQagZq3AGS5BbNPElJ-QF7E8lo', que en este caso sería el identificador de la carpeta bubblesort, se obtiene accediendo a la carpeta en Google Drive (en este caso accediendo a la carpeta bubblesort_db) y en la url se encontraría el identificador:

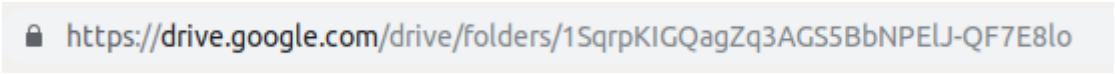

Figura 4.7: Identificador de la carpeta

En la variable 'folder_list' tendríamos una lista con todas las subcarpetas de la carpeta bubblesort_db, por lo tanto una lista de longitud 824. Se puede recorrer esta lista con un bucle para acceder a cada carpeta y a su vez recorrer cada fichero de cada carpeta con otro bucle, con el siguiente código.

```
for folder in folder_list:
  file_list = drive.ListFile({'q': "'"+folder['id']+"' in parents"
      }).GetList()
  for file in file_list:
    if(file['title']=="bubblesort.ARM.elf.trace"):
      download = drive.CreateFile({'id': file['id']})
      download.GetContentFile('trace.log')
      df = createDataframe();
      lifetime, memoryAccess, memoryRead, memoryWrite =
          getInformationProgram()
    if(file['title']=="bubblesort.ARM.elf"):
      download = drive.CreateFile({'id': file['id']})
```

```
        download.GetContentFile('program.elf')
        !objdump -h 'program.elf' > objdump.txt
        text, data, bss, rodata = getSizesProgram()
      if(file['title']=="results.perf"):
        download = drive.CreateFile({'id': file['id']})
        download.GetContentFile('coverage.txt')
        unace, sdc, hang = getCoverage()

      #Write results in a string
      outputToWrite += folder['title']+";"
      for reg in registers:
          outputToWrite += str(lifetime[reg])+";"
      outputToWrite += str(df.shape[0])+";"+str(memoryRead)+";"+str(
         memoryWrite)+";"+str(memoryAccess)+";"+str(text)+";"+str(
         data)+";"+str(bss)+";"+str(rodata)+";"+unace+";"+sdc+";"+
         hang+"\n"
```

Como se puede ver en el código anterior, del fichero 'bubblesort.ARM.elf.trace' obtenemos el tiempo de vida de los registros y los accesos a memoria, del ejecutable 'bubblesort.ARM.elf' obtenemos los tamaños de las secciones text, data, bss y rodata y del fichero 'results.perf' obtenemos los valores unAce, SDC y Hang, todo esto usando las funciones que se han descrito en apartados anteriores.

Estos datos se guardan en un fichero en formato csv y se suben a Google Drive:

```
with open("bubblesortResults.csv", "a") as f:
  f.write(outputToWrite)
  f.close()

fileToUpload = drive.CreateFile({'title': 'bubblesortResults.csv'
   })
fileToUpload.SetContentFile('bubblesortResults.csv')
fileToUpload.Upload()
```

Asi ya se tendrían los datos de todos los programas bubblesort subidos a Google Drive, habría que hacer los mismo pero con los programas Ndes y Dijkstra, que se haría simplemente cambiando el identificador de la carpeta como se ha explicado antes.

Un ejemplo del fichero con los datos de los programas:

1	folder	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	sl	fp	ip	sp	lr
2	b34455	11496	15930	56001	57175	57892	57868	57198	57952	57956	57956	57981	57945	56982	58010	55939
3	252e96	11458	16010	51157	51201	52780	52042	52804	52817	52821	52821	52846	52810	51510	52875	51350
4	b76e50	5909	62137	51257	22298	61497	62322	63301	63314	63318	63318	63343	63307	11408	63372	62321
5	8a4921	5415	15623	51749	51114	52559	51667	52611	52414	52628	52415	52650	52630	51730	52695	51071
6	841e54	195102	25306	76606	112501	195507	195466	195522	195535	195539	195539	195564	195517	317	572	195565
7	0536cd	10959	62536	46210	27844	61894	62718	63701	63714	63718	63718	63743	63707	6358	63772	62721
8	932edb	26108	45898	60662	62096	62972	62235	62999	63012	63016	63016	63041	63005	1117	63070	63027
9	dd18e2	10860	15608	57002	56647	57730	57694	57750	57763	57767	57767	57792	57756	57164	57821	56201
10	8c8e36	11255	15804	56210	56751	57730	57695	57751	57764	57768	57768	57793	57757	57163	57822	56104

Tabla 4.2: Ejemplo datos obtenidos (parte 1)

1	totalInstructions	memoryRead	memoryWrite	memoryAccess	.text	.data	.bss	.rodata	unAce	SDC	Hang
2	58024	10571	404	10975	12220	2392	672	924	76.75	15.25	8
3	52889	10546	390	10936	17916	2392	668	808	78.2777777778	15.5694444444	6.1527777778
4	63386	10449	489	10938	10280	2392	668	808	76.6388888889	18.5416666667	4.8194444444
5	52709	10565	349	10914	11368	2392	668	808	78.4027777778	15.5694444444	6.0277777778
6	195607	107092	25646	132738	18172	2392	668	808	79.4444444444	10.4166666667	10.1388888889
7	63786	10449	489	10938	10344	2392	668	808	76.2916666667	18.6944444444	5.0138888889
8	63084	10546	5241	15787	10344	2392	668	808	77.7916666667	10.7222222222	11.4861111111
9	57835	10548	391	10939	10280	2392	668	808	79	15.25	5.75
10	57836	10546	391	10937	10280	2392	668	808	78.0972222222	15.8611111111	6.0416666667

Tabla 4.3: Ejemplo datos obtenidos (parte 2)

En las tablas de arriba solo se muestran diez filas pero en total habrían 824 para el fichero 'bubblesortResults.csv'. A partir de los datos de los 3 programas crearemos un modelo utilizando redes neuronales para predecir los valores unAce, SDC y Hang.

5 Modelo predictivo

5.1. Introducción

Para desarrollar el modelo predictivo se ha usado una red neuronal artificial creada con keras.

Una red neuronal se compone de varias capas de neuronas (mínimo dos capas, la de entrada y la de salida) conectadas entre sí.

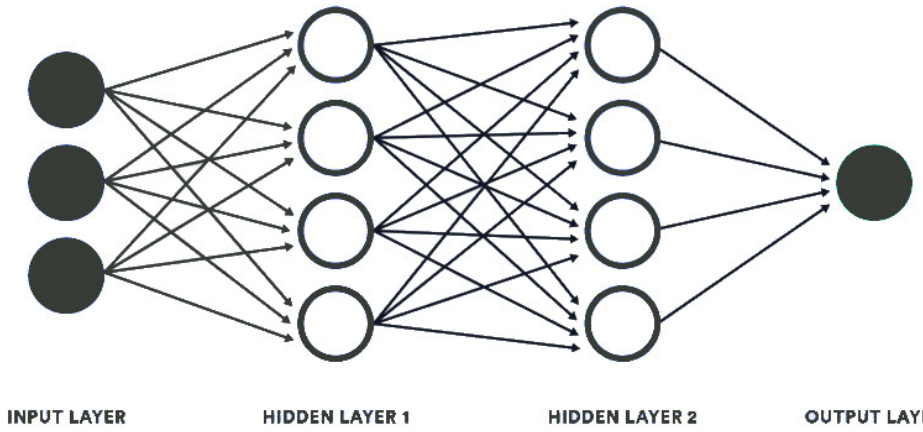

Figura 5.1: Estructura red neuronal

Cada conexión (representada por flechas) tiene asociado un peso que es el valor que ajustará la red durante el proceso de entrenamiento. Para determinar el valor de salida de una neurona se multiplica cada peso por el valor de cada entrada de la neurona y se suman, obteniendo un valor que según la función de activación de la neurona determinará una salida. La salida de las neuronas de la última capa son los valores que ha predicho para una determinada entrada.

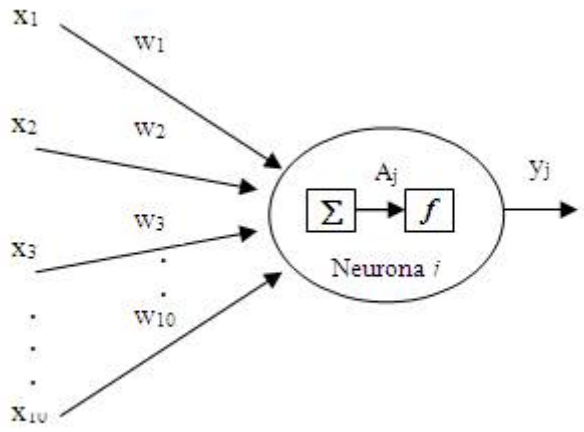

Figura 5.2: Neurona

El proceso de entrenamiento se divide en épocas, cada época se divide a su vez en una serie de iteraciones y en cada iteración se procesan una cantidad de datos según el tamaño del lote (batch size). Si por ejemplo el batch size es de 32, la red neuronal en cada iteración recibiría 32 datos, calcularía el error que se ha producido utilizando una función de pérdida (loss function) y actualizaría los pesos de la red para minimizar el error (backpropagation). Cuando la red neuronal ha procesado todos los datos del conjunto de entrenamiento y actualizado los pesos se dice que se ha completado una época. Keras permite actualizar los pesos de la red de distintas formas utilizando distintos optimizadores. Todo estos parámetros (función de activación, batch size, épocas, etc.) son utilizados por nuestro modelo y más adelante se verá como han sido escogidos.

5.2. Lectura de datos

Primero se descargan los ficheros csv que se han obtenido anteriormente y que están almacenados en Google Drive. El id de los ficheros se obtienen de la url de la opción 'obtener enlace para compartir':

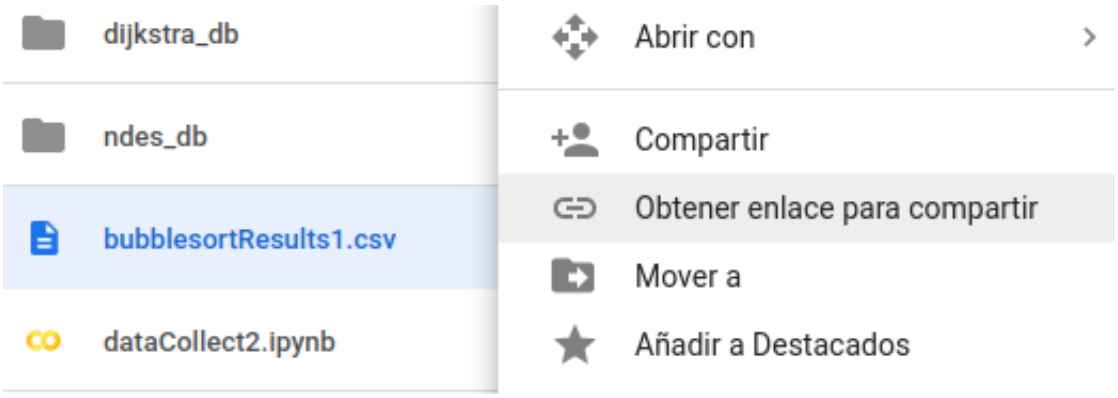

Figura 5.3: Compartir fichero

Y con el siguiente código se descargan los ficheros almacenados en Google Drive:

```
download = drive.CreateFile({'id': '18SnG18K5oT6qvw-6
    Q6JZVYG9pQHI6N7Q'})
download.GetContentFile('dataBubblesort.csv')

download = drive.CreateFile({'id': '1emjNdGJ5fV6K6BSu-
    wwCCUIOBXjMbj4C'})
download.GetContentFile('dataNdes.csv')
```

Una vez descargados se leen utilizando la librería pandas y los juntamos en un único dataset:

```
dfBubblesort = pandas.read_csv("dataBubblesort.csv", sep=";")
dfNdes = pandas.read_csv("dataNdes.csv", sep=";")
df = pandas.concat([dfBubblesort, dfNdes])
```

```
print(df.shape)
```

```
=> (1542, 27)
```

El tamaño del conjunto de datos total es de 1542.

Estos datos se cargan en arrays numpys que son el tipo de datos con el que se trabaja en keras. Se dividen los datos en datos de entrada (x_data) y en etiquetas o datos de salida (y_data).

```
x_data = df[["r0","r1","r2","r3","r4","r5","r6","r7","r8","r9",
             "sl","fp","ip","lr","totalInstructions",
             "memoryRead","memoryWrite","memoryAccess",
             ".text",".data",".bss",".rodata"]].values

y_data = df[["SDC","Hang"]].values
```

Como se puede ver en el código de arriba, solo se almacena en y_data los valores SDC y Hang esto es porque $SDC + Hang + unAce = 100$, por lo que obteniendo dos de estos valores se puede obtener el tercero fácilmente, por este motivo la red tendrá dos neuronas en la capa de salida en vez de tres.

5.3. Estructura del entrenamiento y de la evaluación del modelo

Se divide el conjunto de datos en un 10 % para test y el 90 % restante para la optimización de los parámetros del modelo utilizando validación cruzada (cross validation).

El conjunto de test se utilizará al final, con el modelo definitivo, para evaluar como de bien lo haría nuestro modelo prediciendo datos que nunca ha visto.

Con el otro 90 % del conjunto de datos se irán probando modelos con distintos parámetros y evaluándolos con validación cruzada de 10 iteraciones (kfold de 10), quedándonos con los parámetros que mejor resultados nos den.

El modelo con los parámetros optimizados será el modelo final que evaluaremos con el conjunto de test que se había separado al principio. Este resultado será una aproximación del rendimiento de nuestro modelo en el "mundo real".

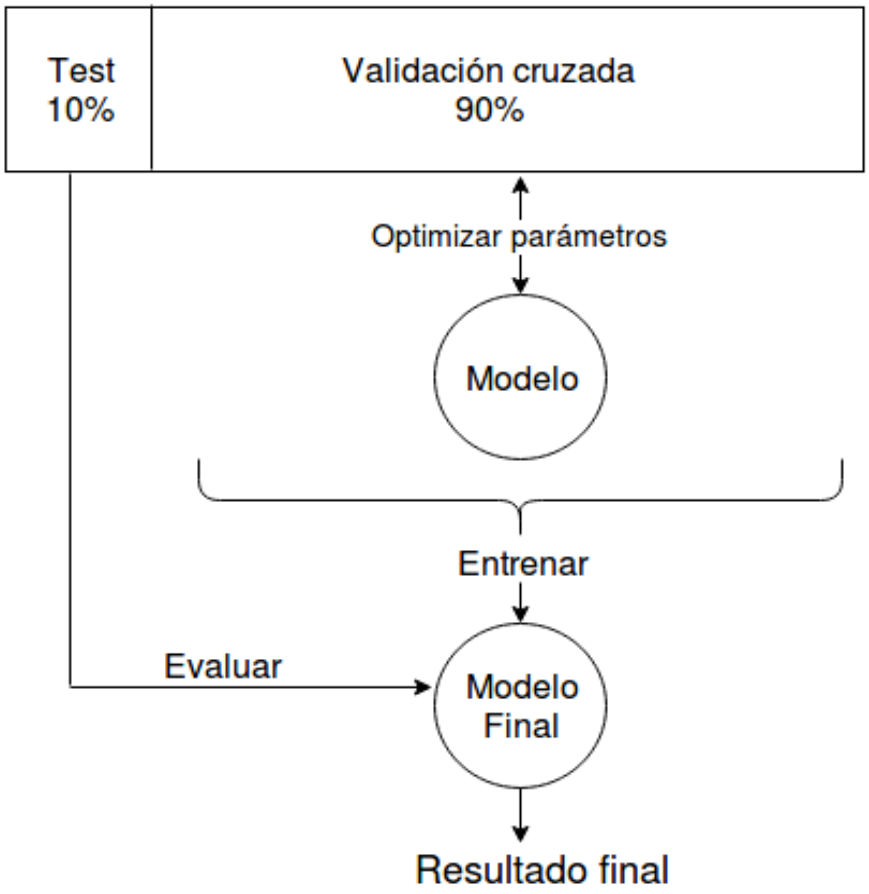

Figura 5.4: Esquema para la obtención del modelo final

5.4. Validación cruzada

En la anterior sección se ha visto la estructura general para entrenar y evaluar el modelo. Aquí se entrará más en detalle en la parte de la optimización de parámetros del modelo utilizando validación cruzada.

La validación cruzada consiste en separar los datos en k particiones (en este caso k sería 10) y realizar k iteraciones, en cada iteración se entrena el modelo con todas las particiones excepto una que se utiliza para evaluar el modelo, y en cada iteración se escogen particiones de entrenamiento y validación distintas. El resultado de la validación

cruzada es la media de los resultados de las k evaluaciones realizadas. Viendo un ejemplo de la validación cruzada con k=4:

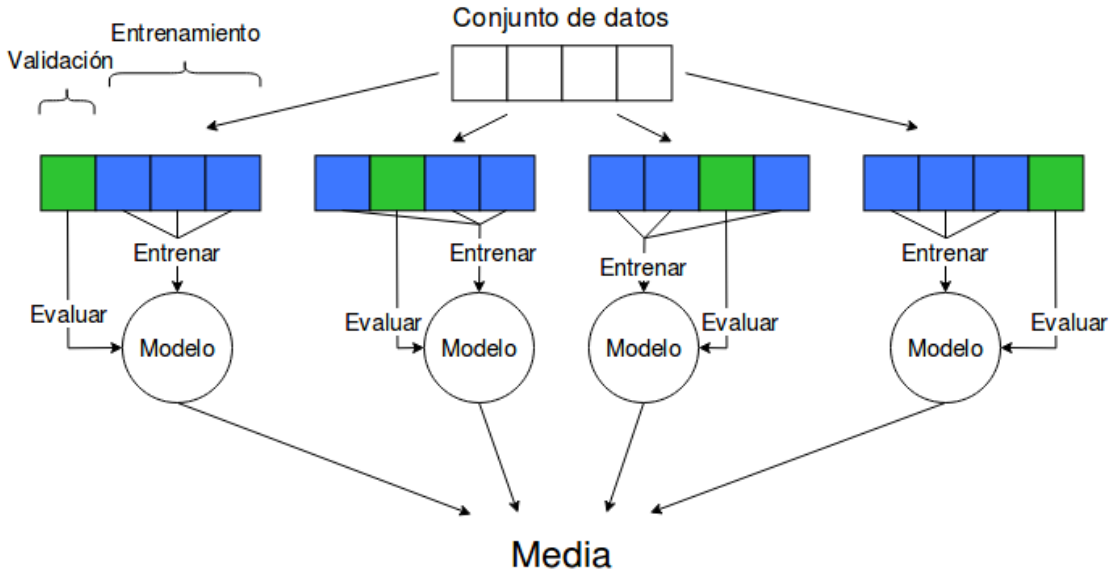

Figura 5.5: Validación cruzada con k=4

La validación cruzada nos permite evaluar de manera más fiable nuestro modelo minimizando los errores de haber escogido para validar y/o entrenar muestras poco representativas.

No hay que confundir la validación con el test. Aunque el modelo no use el conjunto de validación para entrenar, si que se toman decisiones en cuanto a la elección de los parámetros del modelo a partir del resultado del modelo en el conjunto de validación, por lo que se podría decir que el modelo se ve influenciado indirectamente por el conjunto de validación. Por ello, una vez tenemos el modelo final, se utilizará el conjunto de test para obtener el resultado final de nuestro modelo en el "mundo real", pero no se toman decisiones de optimización a partir de este resultado final.

5.4.1. Búsqueda de parámetros óptimos

Para la comparación de parámetros se utiliza la librería sklearn, en concreto las clases GridSearchCV y RandomizedSearchCV. Grid search permite a partir de un diccionario con distintos parámetros realizar todas las combinaciones posibles y devolverte la mejor combinación. Hacer esto puede ser computacionalmente muy costoso, para solucionarlo se pueden realizar combinaciones al azar (RandomizedSearchCV) o en vez de pasarle un diccionario grande a GridSearchCV pasarle varios más pequeños y así optimizar los

parámetros por bloques.

Como se ha comentado anteriormente separamos los datos en un 10 % para test y el 90 % restante para la optimización de los parámetros con validación cruzada:

```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data
    , test_size=0.1, random_state=0, shuffle=True)
```

Después de dividir los datos se normalizan los datos de entrada (las etiquetas no harían falta) con los que se va a trabajar, esto se hace cuando se trabaja con datos en diferentes escalas (por ejemplo edad y sueldo) para que la red neuronal no otorgue mayor importancia al dato con mayor escala y también permitiría a la red aprender más rápido (menos épocas) [11]:

```
scaler = preprocessing.StandardScaler() #Values with mean=0 and
    standard deviation=1

x_trainOpt = scaler.fit_transform(x_train)
```

La normalización aplicada es dejar los datos con un valor medio de 0 y una desviación estándar de 1.

Luego se empaquetaría el modelo utilizando la clase KerasRegressor para poder usarlo junto a la librería sklearn. Keras Regressor recibe como parámetros una función que devuelve un modelo y el número de épocas.

```
model = KerasRegressor(build_fn=nn_modelToOptimize, epochs=150)
```

Esta función que devuelve el modelo y que también tiene que recibir como entrada todos los parámetros que queremos optimizar sería:

```
def nn_modelToOptimize(optimizer='SGD', activation='tanh',
    activation_output='linear', loss='mean_absolute_error',
    init_mode='glorot_uniform', nl=6, nn1=128, nn2=128, nn3=22, nn4
    =22, nn5=64, nn6=44, nn7=11):

  visible = Input(shape=(x_data.shape[1],))

  first = True
  weights = [nn1, nn2, nn3, nn4, nn5, nn6, nn7]
  for i in range(nl):
    if first:
      hidden = Dense(weights[i], activation=activation,
```

```
            kernel_initializer=init_mode) (visible)
        first = False
      else:
        hidden = Dense(weights[i], activation=activation,
            kernel_initializer=init_mode) (hidden)


    output = Dense(2, activation=activation_output,
        kernel_initializer=init_mode) (hidden)

    model = Model(visible, output)
    model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy
        '])

    return model
```

Los parámetros que recibe la función y que se van a optimizar son: el optimizador (optimizer), la función de activación de las capas ocultas (activation), la función de activación de la capa de salida (activation_output), la función de perdida (loss), la forma de inicializar los pesos (init_mode), el número de capas de la red (nl) y el número de neuronas en cada capa (nn1, nn2, nn3, nn4, nn5).

Se crean tantas listas como parámetros se quieran optimizar (en este ejemplo dos listas una para el optimizador y otra para la función de perdida) definiendo en cada lista los distintos tipos de optimizadores y funciones de perdida que se quieran probar. Con estas listas se crea un diccionario, el cual tiene que tener como nombre de cada clave el mismo nombre de la variable de entrada de la función que devuelve el modelo.

```
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adam', 'Adamax']
loss = ['mean_squared_error', 'mean_absolute_error', '
    mean_squared_logarithmic_error']

param_grid = dict(optimizer=optimizer, loss=loss)
```

Por último, para realizar la búsqueda de parámetros se podrían hacer todas las combinaciones posibles o elegir combinaciones al azar. En el ejemplo de arriba habría en cada lista 5 y 3 elementos respectivamente por lo que habrían 15 permutaciones distintas.

Para obtener el resultado de todas las combinaciones de los parámetros utilizamos GridSearchCV. Está clase recibe como entrada el modelo que hemos empaquetado antes, la forma de evaluar que modelo es mejor que en este caso se utiliza el error medio absoluto, el diccionario con los parámetros que queremos probar, el numero de procesos concurrentes que utiliza para realizar el cálculo (al poner -1 utilizaría tantos procesos como núcleos tenga tu procesador) y el valor de la k en la validación cruzada que como

se ha comentado antes se utilizarán 10 particiones.

```
grid = GridSearchCV(estimator=model, scoring='
    neg_mean_absolute_error', param_grid=param_grid, n_jobs=-1, cv
    =10)

grid_result = grid.fit(x_trainOpt, y_train)
```

Luego le pasaríamos los datos de entrada y etiquetas y nos devolvería el modelo que mejor resultados ha dado.

Para obtener el resultado con combinaciones al azar se utiliza la clase RandomizedSearchCV con los mismos parámetros que la clase anterior pero añadiendo el número de combinaciones que se quieren probar (n_iter).

```
grid = RandomizedSearchCV(estimator=model, scoring='
    neg_mean_absolute_error', n_iter=10, param_distributions=
    param_grid, cv=10, n_jobs=-1)
```

Se ha visto un ejemplo para entender como funciona esta optimización de parámetros. Ahora se van a ver las pruebas que se han hecho para obtener unos parámetros óptimos.

5.4.1.1. Número de capas y neuronas por capa

Para elegir la estructura de la red se van a probar con distintas capas y distintas neuronas por capa.

Se va a optimizar por capas, dejando el número de capas fijas y variando la cantidad de neuronas.

```
# numbers of hidden layers
nl = [1]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
```

Resultados de una red neuronal con 1 capa oculta y probando distintas cantidades de neuronas en esa capa:

```
Fitting 10 folds for each of 6 candidates, totalling 60 fits
```

```
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed: 10.9min
   finished
-2.612460 (0.273035) with: {'nn1': 22, 'nl': 1}
-2.634249 (0.306399) with: {'nn1': 44, 'nl': 1}
-2.652237 (0.277614) with: {'nn1': 64, 'nl': 1}
-2.688580 (0.304675) with: {'nn1': 11, 'nl': 1}
-2.745827 (0.299831) with: {'nn1': 128, 'nl': 1}
-3.097432 (0.285675) with: {'nn1': 2, 'nl': 1}
```

Con los parámetros escogidos hay 6 permutaciones posibles que junto a la validación cruzada de 10 particiones da un total de 60 entrenamientos/validaciones a realizar que ha tardado 10.9 minutos.

El mejor resultado se obtiene con una capa oculta y 22 neuronas.

Ahora se van a ver resultados añadiendo más capas.

Resultados de una red neuronal con 2 capas ocultas y probando distintas cantidades de neuronas en esas capas:

```
# numbers of hidden layers
nl = [2]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
nn2=[2, 11, 22, 44, 64, 128]
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
[Parallel(n_jobs=-1)]: Done 360 out of 360 | elapsed: 68.9min
   finished
-2.149143 (0.211140) with: {'nl': 2, 'nn1': 44, 'nn2': 22}
-2.158340 (0.263038) with: {'nl': 2, 'nn1': 128, 'nn2': 11}
-2.186213 (0.270131) with: {'nl': 2, 'nn1': 11, 'nn2': 22}
-2.206546 (0.195512) with: {'nl': 2, 'nn1': 11, 'nn2': 11}
-2.208441 (0.196882) with: {'nl': 2, 'nn1': 44, 'nn2': 44}
...
-2.889305 (0.315951) with: {'nl': 2, 'nn1': 64, 'nn2': 2}
-2.905803 (0.485323) with: {'nl': 2, 'nn1': 128, 'nn2': 2}
-2.989002 (0.423524) with: {'nl': 2, 'nn1': 44, 'nn2': 2}
-3.034757 (0.430447) with: {'nl': 2, 'nn1': 2, 'nn2': 2}
-3.046592 (0.392170) with: {'nl': 2, 'nn1': 11, 'nn2': 2}
```

En este caso solo se han mostrado las 5 mejores y peores combinaciones de parámetros aunque en total habrían 36. Como se puede ver al añadir dos capas el número de combinaciones posibles aumenta considerablemente al igual que el tiempo de ejecución.

Resultados de una red neuronal con 3 capas ocultas y probando distintas cantidades de neuronas en esas capas:

```
# numbers of hidden layers
nl = [3]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
nn2=[2, 11, 22, 44, 64, 128]
nn3=[2, 11, 22, 44, 64, 128]
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
[Parallel(n_jobs=-1)]: Done 360 out of 360 | elapsed: 72.1min
   finished

-1.949744 (0.304564) with: {'nn3': 44, 'nn2': 64, 'nn1': 11, 'nl':
    3}
-1.969511 (0.287901) with: {'nn3': 11, 'nn2': 64, 'nn1': 22, 'nl':
    3}
-1.971960 (0.476335) with: {'nn3': 22, 'nn2': 22, 'nn1': 22, 'nl':
    3}
-1.972546 (0.620381) with: {'nn3': 44, 'nn2': 44, 'nn1': 44, 'nl':
    3}
-1.985240 (0.283518) with: {'nn3': 64, 'nn2': 22, 'nn1': 128, 'nl
   ': 3}
...
-2.916246 (0.677493) with: {'nn3': 2, 'nn2': 128, 'nn1': 44, 'nl':
    3}
-2.953828 (0.493904) with: {'nn3': 2, 'nn2': 64, 'nn1': 22, 'nl':
   3}
-3.144113 (0.297061) with: {'nn3': 2, 'nn2': 2, 'nn1': 64, 'nl':
   3}
-3.586930 (0.554786) with: {'nn3': 2, 'nn2': 44, 'nn1': 2, 'nl':
   3}
-3.596043 (0.578936) with: {'nn3': 2, 'nn2': 22, 'nn1': 2, 'nl':
   3}
```

El número total de combinaciones es de 216 pero se han escogido solo 36 combinaciones al azar ya que sino el tiempo de ejecución aumentaría a las 6 horas.

Resultados de una red neuronal con 4 capas ocultas y probando distintas cantidades de neuronas en esas capas:

```
# numbers of hidden layers
nl = [4]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
```

```
nn2=[2, 11, 22, 44, 64, 128]
nn3=[2, 11, 22, 44, 64, 128]
nn4=[2, 11, 22, 44, 64, 128]
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
[Parallel(n_jobs=-1)]: Done 360 out of 360 | elapsed: 76.3min
   finished

-1.862083 (0.331064) with: {'nn4': 44, 'nn3': 22, 'nn2': 11, 'nn1
   ': 44, 'nl': 4}
-1.881543 (0.595963) with: {'nn4': 22, 'nn3': 44, 'nn2': 64, 'nn1
   ': 64, 'nl': 4}
-1.913202 (0.399498) with: {'nn4': 11, 'nn3': 11, 'nn2': 22, 'nn1
   ': 64, 'nl': 4}
-1.927838 (0.419730) with: {'nn4': 11, 'nn3': 11, 'nn2': 64, 'nn1
   ': 128, 'nl': 4}
-1.995676 (0.783194) with: {'nn4': 22, 'nn3': 44, 'nn2': 64, 'nn1
   ': 128, 'nl': 4}
...
-2.995217 (0.599703) with: {'nn4': 2, 'nn3': 64, 'nn2': 128, 'nn1
   ': 44, 'nl': 4}
-3.067678 (0.432709) with: {'nn4': 2, 'nn3': 44, 'nn2': 11, 'nn1':
    22, 'nl': 4}
-3.156945 (0.365349) with: {'nn4': 2, 'nn3': 64, 'nn2': 22, 'nn1':
    11, 'nl': 4}
-3.178025 (0.611737) with: {'nn4': 2, 'nn3': 128, 'nn2': 128, 'nn1
   ': 128, 'nl': 4}
-3.221217 (0.724434) with: {'nn4': 2, 'nn3': 64, 'nn2': 2, 'nn1':
   11, 'nl': 4}
```

Aquí el número de combinaciones posibles sería de 1296 y tardaría 36 horas así que se han escogido 36 combinaciones al azar.

Resultados de una red neuronal con 5 capas ocultas y probando distintas cantidades de neuronas en esas capas:

```
# numbers of hidden layers
nl = [5]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
nn2=[2, 11, 22, 44, 64, 128]
nn3=[2, 11, 22, 44, 64, 128]
nn4=[2, 11, 22, 44, 64, 128]
nn5=[2, 11, 22, 44, 64, 128]
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
[Parallel(n_jobs=-1)]: Done 360 out of 360 | elapsed: 81.6min
   finished

-1.788653 (0.432356) with: {'nn5': 44, 'nn4': 64, 'nn3': 22, 'nn2
   ': 64, 'nn1': 44, 'nl': 5}
-1.817915 (0.466821) with: {'nn5': 22, 'nn4': 128, 'nn3': 22, 'nn2
   ': 128, 'nn1': 44, 'nl': 5}
-1.921782 (0.370127) with: {'nn5': 22, 'nn4': 11, 'nn3': 22, 'nn2
   ': 128, 'nn1': 128, 'nl': 5}
-1.933658 (0.471119) with: {'nn5': 128, 'nn4': 128, 'nn3': 64, '
   nn2': 64, 'nn1': 44, 'nl': 5}
-1.934345 (0.418070) with: {'nn5': 128, 'nn4': 11, 'nn3': 128, '
   nn2': 44, 'nn1': 11, 'nl': 5}
...
-3.011144 (0.484605) with: {'nn5': 2, 'nn4': 22, 'nn3': 2, 'nn2':
   64, 'nn1': 22, 'nl': 5}
-3.076931 (0.630580) with: {'nn5': 2, 'nn4': 64, 'nn3': 64, 'nn2':
    64, 'nn1': 64, 'nl': 5}
-3.185683 (0.600534) with: {'nn5': 2, 'nn4': 64, 'nn3': 128, 'nn2
   ': 44, 'nn1': 64, 'nl': 5}
-3.197488 (0.465659) with: {'nn5': 2, 'nn4': 128, 'nn3': 22, 'nn2
   ': 44, 'nn1': 11, 'nl': 5}
-3.510690 (0.507116) with: {'nn5': 2, 'nn4': 64, 'nn3': 2, 'nn2':
   44, 'nn1': 22, 'nl': 5}
```

En este caso el número de combinaciones posibles sería de 7776 y tardaría 216 horas así que también se escogen 36 combinaciones al azar.

Resultados de una red neuronal con 6 capas ocultas y probando distintas cantidades de neuronas en esas capas:

```
# numbers of hidden layers
nl = [6]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
nn2=[2, 11, 22, 44, 64, 128]
nn3=[2, 11, 22, 44, 64, 128]
nn4=[2, 11, 22, 44, 64, 128]
nn5=[2, 11, 22, 44, 64, 128]
nn6=[2, 11, 22, 44, 64, 128]
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
[Parallel(n_jobs=-1)]: Done 360 out of 360 | elapsed: 85.6min
   finished
```

```
-1.692845 (0.298587) with: {'nn6': 44, 'nn5': 64, 'nn4': 22, 'nn3
    ': 22, 'nn2': 128, 'nn1': 128, 'nl': 6}
-1.918815 (0.399145) with: {'nn6': 128, 'nn5': 22, 'nn4': 22, 'nn3
    ': 22, 'nn2': 22, 'nn1': 128, 'nl': 6}
-2.012164 (0.341446) with: {'nn6': 128, 'nn5': 11, 'nn4': 128, '
    nn3': 44, 'nn2': 22, 'nn1': 22, 'nl': 6}
-2.017365 (0.509307) with: {'nn6': 11, 'nn5': 64, 'nn4': 11, 'nn3
    ': 128, 'nn2': 22, 'nn1': 128, 'nl': 6}
-2.084895 (0.571322) with: {'nn6': 11, 'nn5': 64, 'nn4': 2, 'nn3':
     44, 'nn2': 44, 'nn1': 64, 'nl': 6}
...
-2.804111 (0.702116) with: {'nn6': 11, 'nn5': 2, 'nn4': 22, 'nn3':
     2, 'nn2': 11, 'nn1': 22, 'nl': 6}
-2.909117 (0.697301) with: {'nn6': 44, 'nn5': 2, 'nn4': 2, 'nn3':
    64, 'nn2': 11, 'nn1': 2, 'nl': 6}
-2.973673 (0.817128) with: {'nn6': 44, 'nn5': 2, 'nn4': 11, 'nn3':
     128, 'nn2': 128, 'nn1': 2, 'nl': 6}
-2.974718 (0.545324) with: {'nn6': 22, 'nn5': 128, 'nn4': 44, 'nn3
    ': 2, 'nn2': 22, 'nn1': 2, 'nl': 6}
-3.860455 (0.767751) with: {'nn6': 2, 'nn5': 22, 'nn4': 44, 'nn3':
     128, 'nn2': 11, 'nn1': 2, 'nl': 6}
```

Resultados de una red neuronal con 7 capas ocultas y probando distintas cantidades de neuronas en esas capas:

```
# numbers of hidden layers
nl = [7]
# neurons in each layer
nn1=[2, 11, 22, 44, 64, 128]
nn2=[2, 11, 22, 44, 64, 128]
nn3=[2, 11, 22, 44, 64, 128]
nn4=[2, 11, 22, 44, 64, 128]
nn5=[2, 11, 22, 44, 64, 128]
nn6=[2, 11, 22, 44, 64, 128]
nn7=[2, 11, 22, 44, 64, 128]
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
[Parallel(n_jobs=-1)]: Done 360 out of 360 | elapsed: 89.2min
    finished

-1.893928 (0.432301) with: {'nn7': 128, 'nn6': 22, 'nn5': 44, 'nn4
    ': 11, 'nn3': 128, 'nn2': 22, 'nn1': 128, 'nl': 7}
-1.916262 (0.562385) with: {'nn7': 44, 'nn6': 128, 'nn5': 64, 'nn4
    ': 44, 'nn3': 44, 'nn2': 128, 'nn1': 44, 'nl': 7}
-1.986767 (0.450292) with: {'nn7': 22, 'nn6': 128, 'nn5': 22, 'nn4
    ': 11, 'nn3': 64, 'nn2': 44, 'nn1': 22, 'nl': 7}
```

```
-1.998152 (0.391680) with: {'nn7': 64, 'nn6': 128, 'nn5': 44, 'nn4
    ': 44, 'nn3': 11, 'nn2': 44, 'nn1': 64, 'nl': 7}
-2.016004 (0.488600) with: {'nn7': 64, 'nn6': 22, 'nn5': 64, 'nn4
    ': 44, 'nn3': 128, 'nn2': 44, 'nn1': 44, 'nl': 7}
...
-2.791787 (0.639480) with: {'nn7': 11, 'nn6': 64, 'nn5': 64, 'nn4
    ': 44, 'nn3': 2, 'nn2': 64, 'nn1': 2, 'nl': 7}
-2.803504 (0.731797) with: {'nn7': 128, 'nn6': 64, 'nn5': 22, 'nn4
    ': 2, 'nn3': 2, 'nn2': 11, 'nn1': 2, 'nl': 7}
-3.081624 (0.700066) with: {'nn7': 2, 'nn6': 128, 'nn5': 64, 'nn4
    ': 64, 'nn3': 11, 'nn2': 128, 'nn1': 64, 'nl': 7}
-3.100566 (0.708637) with: {'nn7': 11, 'nn6': 64, 'nn5': 2, 'nn4':
     11, 'nn3': 11, 'nn2': 22, 'nn1': 2, 'nl': 7}
-4.320603 (0.544898) with: {'nn7': 2, 'nn6': 22, 'nn5': 22, 'nn4':
     2, 'nn3': 128, 'nn2': 128, 'nn1': 2, 'nl': 7}
```

Con 7 capas ocultas los resultados empeoran ligeramente.

Finalmente la estructura del modelo será de 6 capas ocultas con 128 neuronas en la primera capa, 128 en la segunda, 22 en la tercera, 22 en la cuarta, 64 en la quinta y 44 en la sexta. Esta combinación de parámetros es la que ha obtenido un menor error medio absoluto, exactamente un 1.692845

El error medio absoluto que se ve aquí es la diferencia entre las predicciones que ha realizado el modelo y el valor real que debería haber predicho utilizando los datos del conjunto de validación y dividiendo la suma absoluta de los errores entre el número de elementos del conjunto de validación para obtener el error medio que es el valor que nos dirá como de bien predice el modelo, cuanto más bajo sea este valor mejor.

Como se puede ver los valores de error absoluto que se devuelven están en negativo esto es porque la función ordena de mayor a menor los resultados considerando un mayor valor como mejor pero en el caso del error cuanto menor sea este mejor es el resultado por lo tanto se utiliza el negado del error medio absoluto para que estén correctamente ordenados los resultados.

5.4.1.2. Batch size, función de perdida y inicialización de los pesos

Igual que antes, se ponen todos los parámetros que se quieren probar en un diccionario y se ejecuta en este caso un grid search.

```
batch_sizeT = [16, 32, 64, 128]
loss = ['mean_squared_error', 'mean_absolute_error', '
    mean_squared_logarithmic_error', '
```

```
    mean_absolute_percentage_error', 'logcosh']
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', '
    glorot_normal', 'glorot_uniform', 'he_normal', 'he_uniform']
```

```
Fitting 10 folds for each of 160 candidates, totalling 1600 fits
[Parallel(n_jobs=-1)]: Done 1600 out of 1600 | elapsed: 361.2min
    finished

-1.433648 (0.307590) with: {'batch_size': 16, 'init_mode': '
    he_normal', 'loss': 'mean_absolute_error'}
-1.516588 (0.341323) with: {'batch_size': 64, 'init_mode': '
    he_normal', 'loss': 'mean_absolute_error'}
-1.519136 (0.365936) with: {'batch_size': 16, 'init_mode': '
    he_normal', 'loss': 'logcosh'}
-1.558980 (0.528033) with: {'batch_size': 16, 'init_mode': '
    he_uniform', 'loss': 'mean_absolute_error'}
-1.599890 (0.495651) with: {'batch_size': 16, 'init_mode': '
    he_uniform', 'loss': 'logcosh'}
...
-10.900354 (0.058840) with: {'batch_size': 64, 'init_mode': 'zero
    ', 'loss': 'mean_squared_logarithmic_error'}
-10.900354 (0.058840) with: {'batch_size': 32, 'init_mode': 'zero
    ', 'loss': 'mean_squared_logarithmic_error'}
-10.900354 (0.058840) with: {'batch_size': 16, 'init_mode': 'zero
    ', 'loss': 'mean_squared_logarithmic_error'}
-11.948235 (10.682711) with: {'batch_size': 16, 'init_mode': '
    lecun_uniform', 'loss': 'mean_absolute_percentage_error'}
-12.212963 (11.166714) with: {'batch_size': 16, 'init_mode': '
    he_uniform', 'loss': 'mean_absolute_percentage_error'}
```

En total salen 160 combinaciones distintas con una duración de 361.1 minutos. De los resultados se puede ver que es una mala idea inicializar los pesos a cero o elegir como función de perdida a 'mean_absolute_percentage_error'. En cambio los parámetros que mejor resultados dan son un batch size de 16, inicializar los pesos con la función 'he_normal' y utilizar como función de perdida 'mean_absolute_error'. Para probar los otros parámetros dejaremos en el modelo estos tres parámetros por defecto.

5.4.1.3. Funciones de activación y optimizadores

Se van a probar distintas combinaciones de funciones de activación y optimizadores. Las funciones de activaciones vamos a dividirlas en funciones de activación para las capas ocultas y función de activación para la capa de salida. Como las etiquetas están en un rango de 0 a 100 (al ser probabilidades) las funciones de activación como 'sigmoid' o

'tanh' no servirían al estar su salida en un rango de 0 a 1 y de -1 a 1 respectivamente. Para probar estas últimas funciones de activación en la capa de salida se tendrían que dividir las etiquetas entre 100 para dejarlas en un rango de 0 a 1.

Por ello primero se van a probar en la capa de salida las funciones de activación que devuelven valores en el rango de 0 a 100 y después se van a modificar las etiquetas para que estén en el rango 0 a 1 y probar las funciones de activación de la capa de salida restantes.

```
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adam', 'Adamax']
activation = ['relu', 'tanh', 'sigmoid', 'elu', 'selu', '
    hard_sigmoid']
activation_output = ['linear', 'relu']
```

```
Fitting 10 folds for each of 60 candidates, totalling 600 fits
[Parallel(n_jobs=-1)]: Done 600 out of 600 | elapsed: 484.0min
    finished

-1.341684 (0.336686) with: {'activation': 'selu', '
    activation_output': 'linear', 'optimizer': 'Adamax'}
-1.345840 (0.303571) with: {'activation': 'selu', '
    activation_output': 'linear', 'optimizer': 'RMSprop'}
-1.355310 (0.280410) with: {'activation': 'selu', '
    activation_output': 'relu', 'optimizer': 'RMSprop'}
-1.371299 (0.274406) with: {'activation': 'relu', '
    activation_output': 'linear', 'optimizer': 'Adamax'}
-1.376356 (0.432091) with: {'activation': 'elu', '
    activation_output': 'linear', 'optimizer': 'RMSprop'}
...
-7.335668 (3.151692) with: {'activation': 'hard_sigmoid', '
    activation_output': 'relu', 'optimizer': 'Adamax'}
-7.484364 (2.685998) with: {'activation': 'hard_sigmoid', '
    activation_output': 'relu', 'optimizer': 'SGD'}
-7.831220 (2.756525) with: {'activation': 'hard_sigmoid', '
    activation_output': 'relu', 'optimizer': 'Adam'}
-8.127785 (3.126297) with: {'activation': 'hard_sigmoid', '
    activation_output': 'relu', 'optimizer': 'RMSprop'}
-8.769912 (1.994023) with: {'activation': 'sigmoid', '
    activation_output': 'relu', 'optimizer': 'SGD'}
```

Y ahora se van a probar los mismos parámetros, cambiando solo las funciones de activación de la capa de salida y dividiendo las etiquetas entre 100.

```
y_data = y_data/100.0

```

```
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adam', 'Adamax']
activation = ['relu', 'tanh', 'sigmoid', 'elu', 'selu', '
    hard_sigmoid']
activation_output = ['tanh', 'sigmoid']
```

```
Fitting 10 folds for each of 60 candidates, totalling 600 fits
[Parallel(n_jobs=-1)]: Done 600 out of 600 | elapsed: 335.1min
    finished

-0.011153 (0.002503) with: {'activation': 'relu', '
    activation_output': 'tanh', 'optimizer': 'Adamax'}
-0.012177 (0.003793) with: {'activation': 'relu', '
    activation_output': 'tanh', 'optimizer': 'Adagrad'}
-0.012257 (0.002745) with: {'activation': 'relu', '
    activation_output': 'sigmoid', 'optimizer': 'Adamax'}
-0.013823 (0.002706) with: {'activation': 'tanh', '
    activation_output': 'sigmoid', 'optimizer': 'Adamax'}
-0.014069 (0.002865) with: {'activation': 'elu', '
    activation_output': 'sigmoid', 'optimizer': 'RMSprop'}
...
-0.032245 (0.011416) with: {'activation': 'hard_sigmoid', '
    activation_output': 'tanh', 'optimizer': 'RMSprop'}
-0.044759 (0.004061) with: {'activation': 'hard_sigmoid', '
    activation_output': 'sigmoid', 'optimizer': 'SGD'}
-0.044775 (0.004072) with: {'activation': 'sigmoid', '
    activation_output': 'sigmoid', 'optimizer': 'SGD'}
-0.057230 (0.011308) with: {'activation': 'sigmoid', '
    activation_output': 'tanh', 'optimizer': 'SGD'}
-0.060027 (0.018258) with: {'activation': 'hard_sigmoid', '
    activation_output': 'tanh', 'optimizer': 'SGD'}
```

Como es normal al dividir las etiquetas entre 100 el error absoluto es mucho menor, así que los valores de error absoluto que salen habría que multiplicarlos por 100 para compararlos con los demás. Haciendo esto se ven que los mejores parámetros son la función de activación 'relu' en las capas ocultas, la función de activación 'tanh' en la capa de salida y el optimizador 'Adamax'.

5.4.1.4. Épocas

Para seleccionar el número de épocas se va a realizar una validación cruzada de 10 particiones con el siguiente modelo (con los parámetros que hemos optimizado antes):

```
def nn_model():
  visible = Input(shape=(x_data.shape[1],))
```

```
3
4     hidden = Dense(128, activation='relu', kernel_initializer='
          he_normal') (visible)
5     hidden = Dense(128, activation='relu', kernel_initializer='
          he_normal') (hidden)
6     hidden = Dense(22, activation='relu', kernel_initializer='
          he_normal') (hidden)
7     hidden = Dense(22, activation='relu', kernel_initializer='
          he_normal') (hidden)
8     hidden = Dense(64, activation='relu', kernel_initializer='
          he_normal') (hidden)
9     hidden = Dense(44, activation='relu', kernel_initializer='
          he_normal') (hidden)
10
11    output = Dense(2, activation='tanh', kernel_initializer='
          he_normal') (hidden)
12
13    model = Model(visible, output)
14    model.compile(optimizer='Adamax', loss='mean_absolute_error',
          metrics=['accuracy'])
15
16    return model
```

Se va a entrenar el modelo con 1000 épocas para ver si en algún momento deja de aprender la red o si se produce sobreentrenamiento y así ajustar el número de épocas.

Al realizar validación cruzada de 10 particiones se obtienen 10 resultados distintos para cada partición por lo que se va a mostrar una gráfica con la media de los resultados:

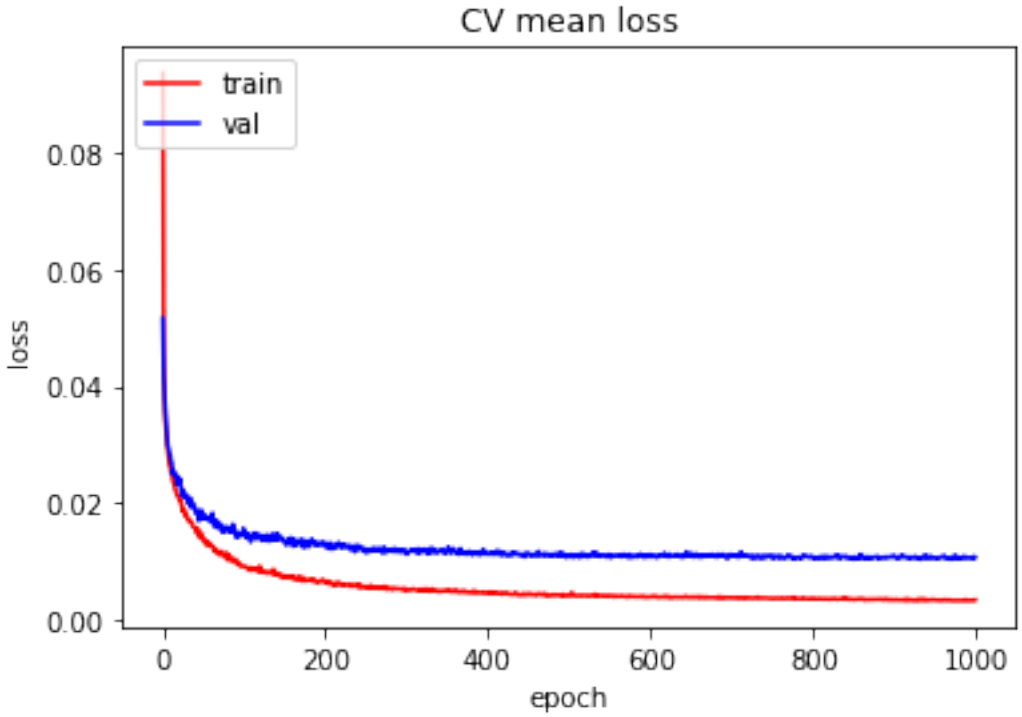

Figura 5.6: 1000 épocas, validación cruzada con k=10

El eje abscisas serían las épocas, el eje de ordenadas el error. Hay que recordar que al haber elegido como función de activación de la capa de salida a 'tanh' las etiquetas se han dividido entre 100 por eso el error en la primera época es de "solamente"0.05.

En la gráfica la linea azul es el error del conjunto de validación en cada época y la roja el error del conjunto de entrenamiento. Lo importante es reducir el error de validación lo máximo posible.

Para ver mas o menos en que época el error de validación deja de bajar se va a mostrar la misma gráfica pero acortando ligeramente el eje de abscisas para que se vea mejor:

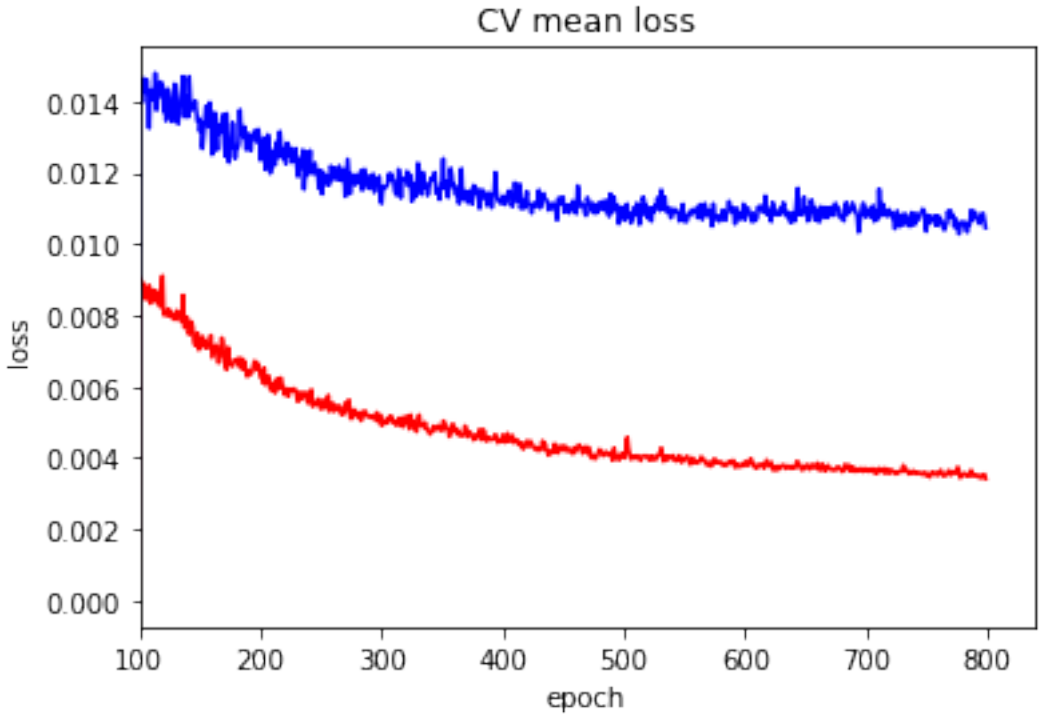

Figura 5.7: Misma gráfica, mostrando desde la época 100 a la 800

Aproximadamente a partir de la época 400 el error de validación permanece constante.

5.5. Modelo final

En la anterior sección se han determinado los parámetros del modelo:

nl	nn1	nn2	nn3	nn4	nn5	nn6
6	128	128	22	22	64	44

Batch size	Funcion de pérdida	Inicialización pesos	Optimizador	Función Activación Capas ocultas	Función Activación Capa de salida	Épocas
16	mean absolute error	he_normal	adamax	relu	tanh	400

Tabla 5.1: Parámetros escogidos para el modelo

Para obtener el modelo final lo entrenamos con el 90 % de datos que habíamos separado anteriormente (figura 5.4).

Primero normalizamos los datos y luego entrenamos el modelo:

```
scaler = preprocessing.StandardScaler() #Values with mean=0 and
    standard deviation=1

x_trainFinal = scaler.fit_transform(x_train)
x_testFinal = scaler.transform(x_test)

model = nn_model()
model.fit(x=x_trainFinal, y=y_train, batch_size=16, epochs=400,
    verbose=2)
```

5.6. Evaluación del modelo

Para saber como de bien predice el modelo vamos a calcular el error medio absoluto con el conjunto de test que corresponde con el 10 % de los datos que nunca se han utilizado ni para entrenar ni para validar (figura 5.4).

```
print(np.abs(y_test-model.predict(x_testFinal)).mean(axis=0))
```

```
[0.00762056 0.00827165]
```

Recordemos que el modelo predice dos valores: SDC y hang. El error medio que comete para SDC es de 0.0076 y para hang de 0.0082. Estos valores son bastante buenos, para compararlos mejor se va a mostrar una tabla comparando el valor real y el valor que se ha predicho de los cinco primeros datos del conjunto de test:

Valor Real (SDC)	Valor predicho (SDC)
15.91 %	15.86 %
19.52 %	18.44 %
16.72 %	16.07 %
15.09 %	15.62 %
0.44 %	0.33 %

Tabla 5.2: Comparativa predicciones (SDC)

Valor Real (hang)	Valor predicho (hang)
6.06 %	5.90 %
5.09 %	4.86 %
5.80 %	5.87 %
5.93 %	5.90 %
20.47 %	21.36 %

Tabla 5.3: Comparativa predicciones (hang)

Los valores de la tabla se han multiplicado por 100 para expresarlos en porcentaje y que se visualicen mejor. Los errores medios serían de 0.76 en SDC y 0.82 para hang si los expresamos en esta escala.

5.7. Otras pruebas

5.7.1. Evaluando el modelo con otro tipo de datos

El conjunto de datos que se ha utilizado hasta ahora estaba formado por información sacada de programas que implementaban los algoritmo bubblesort y ndes como se ha comentado en secciones anteriores. Por lo tanto la información sacada del programa que implementaba el algoritmo dijkstra no se ha utilizado aún. Con esta información queremos comprobar si el modelo predeciría correctamente los valores de la tolerancia a fallo de otros tipos de programas con cuya información nunca ha sido entrenado.

```
1
2  download = drive.CreateFile({'id': '1
       UCMwvHPk27JxErMzAr0DMcJCSz8U8Awq'})
3  download.GetContentFile('dataDijkstra.csv')
4
5  dfDijkstra = pd.read_csv("dataDijkstra.csv", sep=";")
6
7  x_test_dijkstra = dfDijkstra[["r0","r1","r2","r3","r4","r5","r6","
       r7","r8","r9",
8                "sl","fp","ip","lr","totalInstructions",
9                "memoryRead","memoryWrite","memoryAccess",
10               ".text",".data",".bss",".rodata"]].values
11
12 y_test_dijkstra = dfDijkstra[["SDC","Hang"]].values
13
14 y_test_dijkstra = y_test_dijkstra/100.0
```

```
15
16 print(np.abs(y_test_dijkstra-model.predict(x_test_dijkstra_norm)).
       mean(axis=0))
```

```
[0.10776694 0.12945275]
```

El error medio que comete para SDC es de 0.1077 y para hang de 0.1294 que si los expresamos en su escala correspondiente sería de 10.77 y de 12.94 respectivamente. Estos valores son muy altos, sobre todo si los comparamos con los resultados anteriores, por lo tanto nuestro modelo no predeciría correctamente la tolerancia a fallos de otro tipo de programas.

Es probable que esto sea debido a que nuestro conjunto de datos era reducido y que solo ha sido entrenado con dos tipos de programas.

5.7.2. ¿Qué información es la más importante para predecir la tolerancia a fallos?

La información que recibe el modelo son los tiempos de vida de los registros, los accesos a memoria y el tamaño de las cabeceras, a partir de esa información devuelve unas predicciones. Para saber que información es la más relevante para las predicciones del modelo se va a entrenar otra vez el modelo pero modificando los datos con los que se entrena. Se va a poner a 0 una columna del conjunto de datos por ejemplo el tiempo de vida del registro r1 y se entrenará el modelo, si el error que comete el modelo después de haber sido entrenado sin utilizar el tiempo de vida del registro r1 es elevado significa que esa información era importante para el modelo. Se hará esto para todas las columnas del conjunto de datos y se ordenaran según el error que cometa el modelo y así se sabrá que datos son más importantes.

El código que realiza esto es el siguiente:

```
1  #Which inputs are more relevant
2  headers = ["r0","r1","r2","r3","r4","r5","r6","r7","r8","r9",
3             "sl","fp","ip","lr","totalInstructions",
4             "memoryRead","memoryWrite","memoryAccess",
5             ".text",".data",".bss",".rodata"]
6  results_inputs = []
7  for i in range(x_data.shape[1]):
8
9    scaler = preprocessing.StandardScaler()
10   x_train_zero = x_train.copy()
11   x_test_zero = x_test.copy()
```

```
12      x_train_zero[:,i] = 0
13      x_test_zero[:,i] = 0
14      x_trainOneToZero = scaler.fit_transform(x_train_zero)
15      x_testOneToZero = scaler.transform(x_test_zero)
16
17      model = nn_model()
18      model.fit(x=x_trainOneToZero, y=y_train, epochs=400, batch_size
            =16, verbose=2)
19
20      error = np.abs(y_test-model.predict(x_testOneToZero)).mean(axis
            =0)
21      results_inputs.append((headers[i], error))
22
23  sorted_list = results_inputs.copy()
24  sorted_list.sort(reverse=True, key=lambda x : x[1][0]+x[1][1])
```

Al final se obtiene una lista con los datos ordenados de más importantes a menos:

```
[('.text', array([0.01613284, 0.01617962])),
 ('.rodata', array([0.00948478, 0.0095155 ])),
 ('.data', array([0.00866913, 0.00891135])),
 ('r3', array([0.00827907, 0.00922129])),
 ('fp', array([0.00878605, 0.00839737])),
 ('sl', array([0.00872225, 0.00841885])),
 ('r1', array([0.00862888, 0.00790773])),
 ('r4', array([0.00837249, 0.00814616])),
 ('r6', array([0.00850984, 0.00779421])),
 ('r8', array([0.00811555, 0.00798034])),
 ('lr', array([0.00801064, 0.00803852])),
 ('r7', array([0.00797845, 0.00803319])),
 ('.bss', array([0.00792085, 0.00782646])),
 ('ip', array([0.0077841 , 0.00746649])),
 ('r0', array([0.00762029, 0.00748516])),
 ('memoryRead', array([0.00749416, 0.0075207 ])),
 ('memoryAccess', array([0.00727772, 0.00768691])),
 ('r5', array([0.00700997, 0.00729335])),
 ('r9', array([0.00713618, 0.00710026])),
 ('memoryWrite', array([0.00702708, 0.00705298])),
 ('totalInstructions', array([0.0070219 , 0.00692517])),
 ('r2', array([0.00640232, 0.00643851]))]
```

La lista esta formada por tuplas, el primer elemento de la tupla es el dato el cual se ha puesto a cero al entrenar el modelo y el segundo los valores SDC y hang. Como se puede ver el tamaño de la sección .text sería el dato más importante ya que cuando el modelo se entrena sin conocer ese dato el error en su predicción es el que más aumenta.

Ahora se va a hacer otra prueba, se van a agrupar los datos en tres grupos: tiempo de vida de los registros, accesos a memoria y tamaño de las secciones. Se va a entrenar el modelo tres veces poniendo a cero los datos de un grupo distinto cada vez para ver cual de los tres grupos es más importante, ya que antes se ha comprobado cada dato individualmente pero es posible que hayan relaciones entre ellos que hagan que un dato sea importante junto a otro, pero como sería inviable computacionalmente probar todas las combinaciones posibles solo se van a probar estas tres combinaciones.

```
[('Tamaño de las secciones del ejecutable', array([0.01344219,
   0.01396298])),
 ('Tiempo de vida de los registros', array([0.01119967,
    0.01045413])),
 ('Accesos a memoria', array([0.00836928, 0.00787594]))]
```

Haciendo esta prueba se puede ver que el tamaño de las secciones del ejecutable es la información más importante, ya que eliminando esta información es cuando más aumenta el error, seguido de el tiempo de vida de los registros y por último los accesos a memoria.

Por último se va a hacer una prueba similar. Se va a entrenar el modelo también tres veces pero poniendo a cero los datos de dos grupos distintos cada vez. En esta prueba se verá

```
[('Tiempo de vida de los registros', array([0.0150907 ,
   0.01489767])),
 ('Tamaño de las secciones', array([0.02061594, 0.01790487])),
 ('Accesos a memoria', array([0.02898179, 0.02778473]))]
```

De la prueba de arriba se ve que entrenando el modelo solo con los tiempos de vida de los registros se obtienen mejores resultados que entrenando el modelo solo con el tamaño de las secciones o con solo los accesos a memoria ya que se obtiene un modelo que comete un menor error.

6 Conclusiones

En este trabajo se ha visto como se ha obtenido información de ejecutables que puede ser usada para predecir la tolerancia a fallos de estos programas sin tener que ejecutar costosas campañas de inyección de fallos.

También se ha visto que tanto el tiempo de vida de los registros como el tamaño de las secciones del ejecutable son bastante importantes, a diferencia de los accesos a memoria que no lo son tanto.

No se ha podido obtener un modelo que sirva para predecir esta tolerancia a fallos en programas distintos aunque posiblemente sea debido a que no se ha utilizado un conjunto de datos ni muy grande ni muy variado. Por lo tanto una posible continuación de este trabajo sería el obtener más datos con los que entrenar un modelo para predecir la tolerancia a fallos en programas distintos.

Bibliografía

[1] José Isaza-González. Aportaciones a la tolerancia a fallos en microprocesadores bajo efectos de la radiación, 2018.

[2] Simulador OVPSim. http://www.ovpworld.org/technology_ovpsim.

[3] Leonardo Maria Reyneri, Alejandro Serrano-Cases, Yolanda Morilla, Sergio Cuenca-Asensi, and Antonio Martínez-Álvarez. A compact model to evaluate the effects of high level c++ code hardening in radiation environments. *Electronics*, 8(6), 2019.

[4] Google Colab. https://colab.research.google.com/notebooks/basic_features_overview.ipynb.

[5] PyDrive. https://pythonhosted.org/PyDrive/.

[6] Documentación de keras. https://keras.io/.

[7] sklearn. https://scikit-learn.org/stable/.

[8] Expresiones regulares python. https://docs.python.org/2/library/re.html.

[9] Pandas. https://pandas.pydata.org/.

[10] Documentación de arm. http://infocenter.arm.com/help/index.jsp.

[11] Urvashi Jaitley. Why Data Normalization is necessary for Machine Learning models. https://medium.com/@urvashilluniya/why-data-normalization-is-necessary-for-machine-learning-models-681b65a05029, 2018.

[12] C compiler. Memory map. Program in RAM. http://www.ele.uva.es/~jesus/hardware_empotrado/Compiler.pdf, 2019.

[13] Christian Tenllado y Luis Piñuel. Fundamentos de Computadores Manual de Laboratorio. https://eprints.ucm.es/27330/1/FC_ManualLaboratorio.pdf.

[14] Jason Brownlee. How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras. https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/, 2016.

[15] Jason Brownlee. What is the Difference Between Test and Validation Datasets? https://machinelearningmastery.com/difference-test-validation-datasets/, 2017.

[16] Mat Buckland and Mark Collins. *AI Techniques for Game Programming*. Premier Press, 2002.

[17] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[18] Muhammad Ryan. A Simple Way to Know How Important Your Input is in Neural Network. https://medium.com/datadriveninvestor/a-simple-way-to-know-how-important-your-input-is-in-neural-network-86cbae0d3689, 2018.